Filas de Prioridades

Prof. Barbara de Melo Quintela
Prof. José Jerônimo Camata
Prof. Marcelo Caniato
camata@ice.ufjf.br
marcelo.caniato@ice.ufjf.br

Tópicos

1. Introdução
2. Heap Esquerdistas
3. Heap Binomial

Filas de Prioridades - Introdução

- Necessário em aplicações que necessitem determinar repetidas vezes o dado de maior (ou menor) prioridade.
- Exemplos:
  - Sistemas operacionais usam filas de prioridades para escalonar as tarefas no processador.
  - Sistemas de gerenciamento de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Filas de Prioridades

- Pode-se definir uma lista de prioridades como uma tabela na qual a cada um de seus dados está associado a uma prioridade.
- Pode ser implementadas por:
  - Lista não ordenada ou ordenadas (menos eficiente)
  - Heap (mais eficiente)

Implementação por Heap

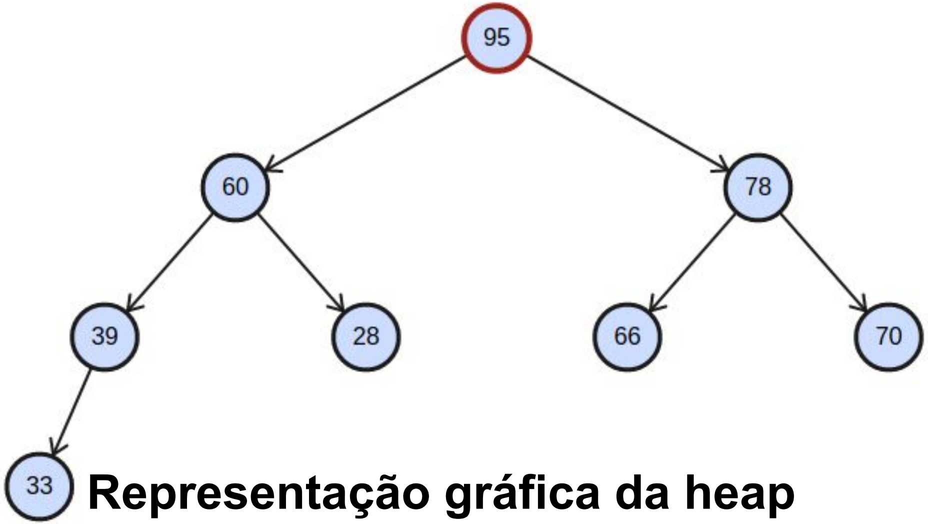
- Uma ***Heap de Máximo*** é um arranjo linear A composta de registros com chaves $A_1, A_2, \ldots, A_n$, satisfazendo a seguinte propriedade:

$$A_i \le A_{\lfloor i/2 \rfloor}$$

para todo elemento exceto a raiz A_1.

Lista de prioridades:
95 60 78 39 28 66 70 33

- Para todo índice i, diremos que
  - $\lfloor i/2 \rfloor$ é o pai do índice i,
  - $2i$ é o filho esquerdo de i,
  - $2i+1$ é o filho direito de i
- O arranjo adquire uma estrutura de árvore binária quase completa e seus elementos, identificados pelos índices 1 a n, passam a ser chamados nós.



Representação gráfica da heap

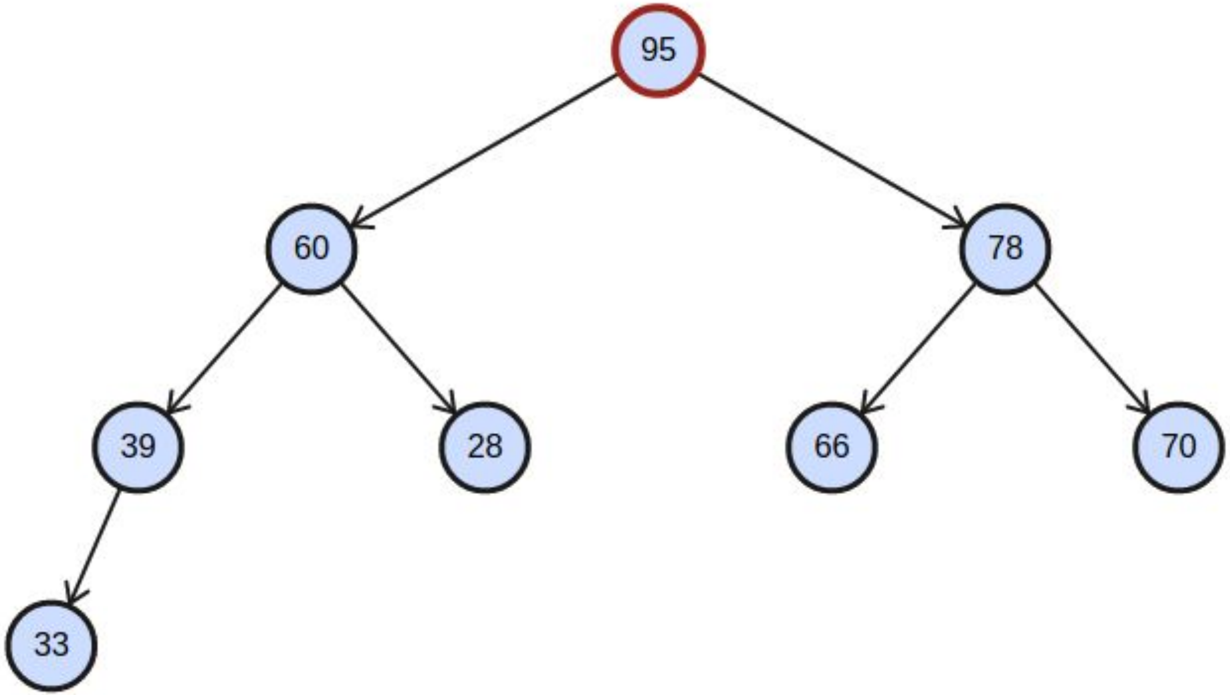
ufjf

Filas de Prioridades

- **Operações Básicas**:
  - ***Seleção*** do elemento de maior (ou menor) prioridade
  - ***Inserção*** de um novo elemento
  - **Remoção** do elemento de maior (ou menor) prioridade.
  - **Alteração** da prioridade de um determinado elemento

Operação de Seleção

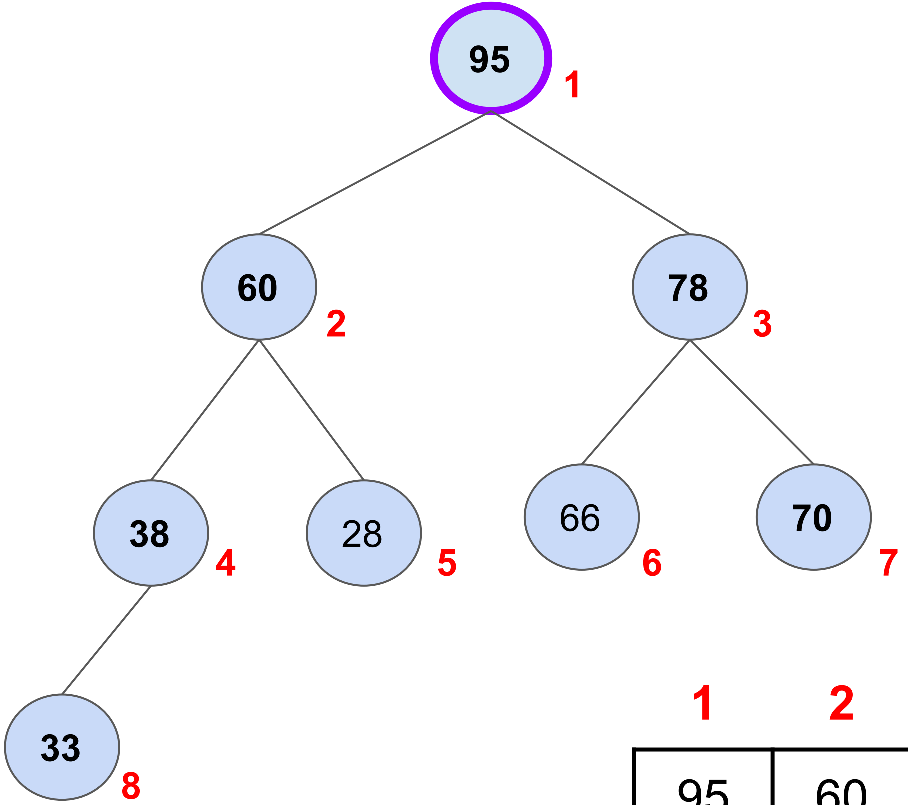
- Basta retornar a posição A[1] do arranjo.



Operação: Alteração de Prioridades

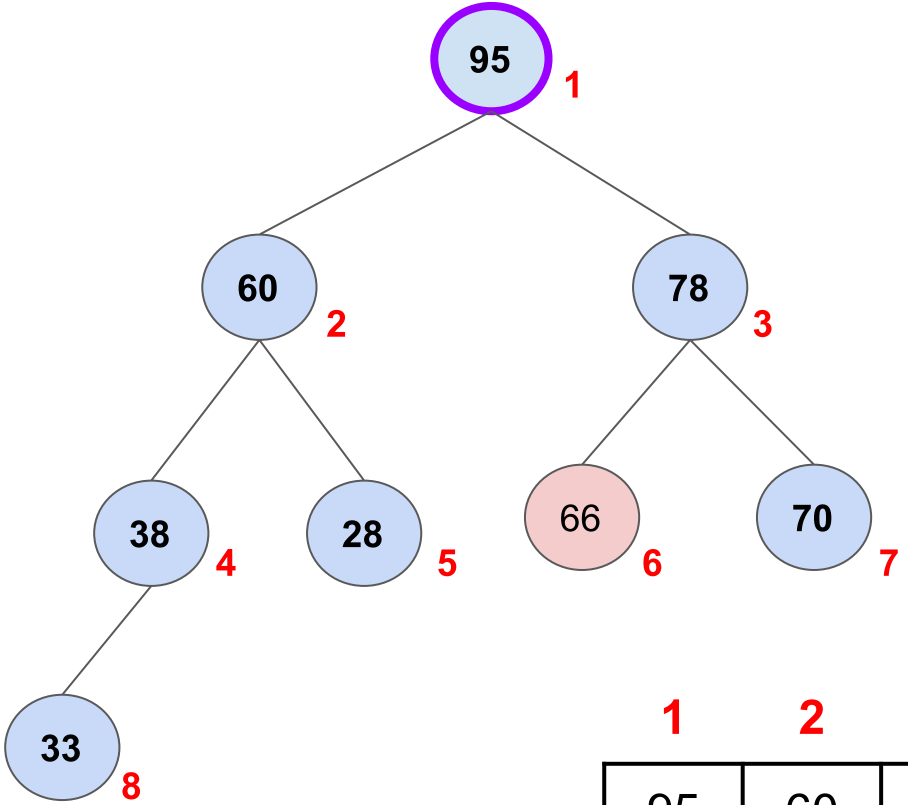
- Na alteração da prioridade de um nó, é necessário reorganizar a heap para que ela respeite as prioridades:
  - registro que tem sua **prioridade aumentada** precisa "**subir**" na heap
  - registro que tem sua **prioridade diminuída** precisa "**descer**"na heap

Exemplo: 1 Alterar a prioridade do nó 6 de 66 para 98



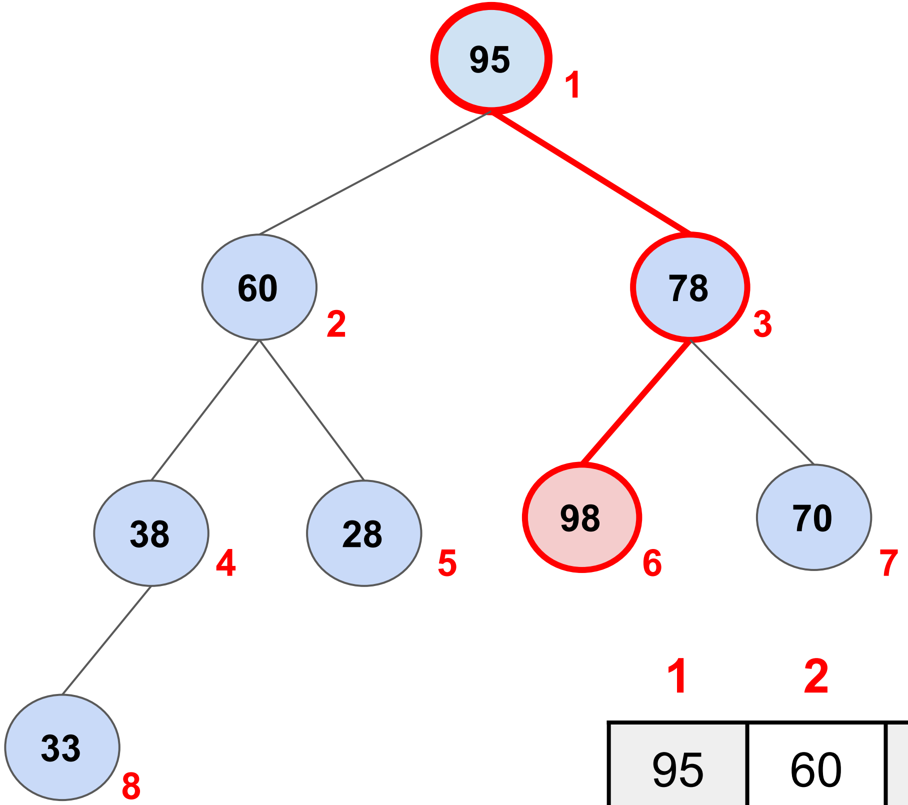
1	2	3	4	5	6	7	8
95	60	78	38	28	66	70	33

Exemplo 1: Alterar a prioridade do nó 6 de 66 para 98



1	2	3	4	5	6	7	8
95	60	78	38	28	66	70	33

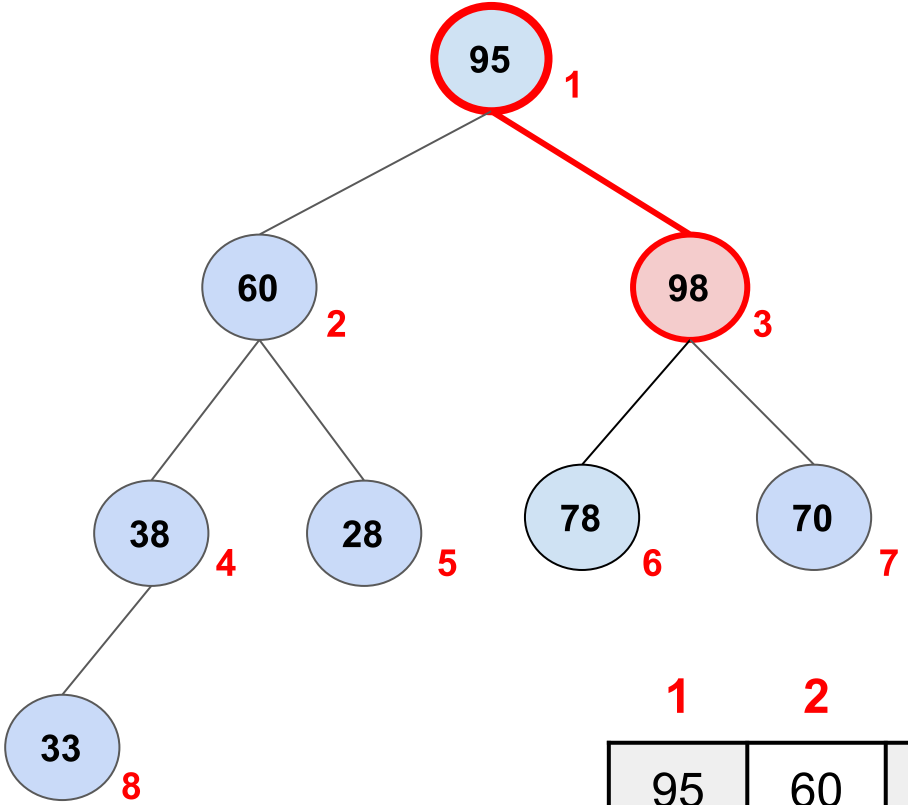
Exemplo 1: Alterar a prioridade do nó 6 de 66 para 98



Apenas o ramo que vai do nó atualizado até a raiz é afetado – o restante da árvore permanece inalterado

1	2	3	4	5	6	7	8
95	60	78	38	28	98	70	33

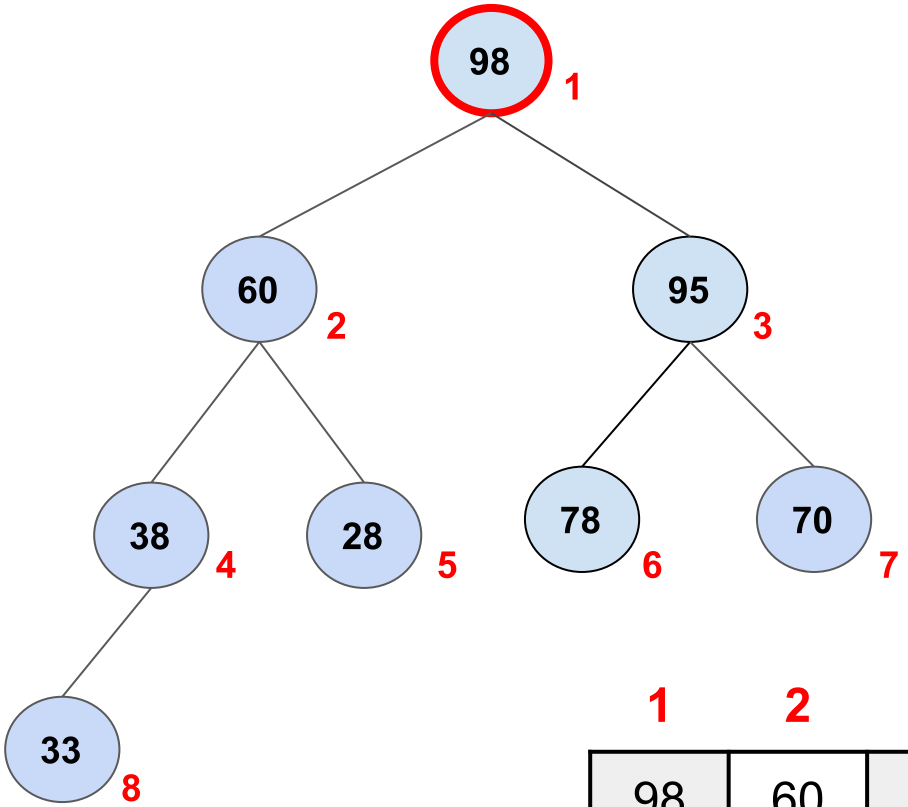
Exemplo 1: Alterar a prioridade do nó 6 de 66 para 98



"Subir" elemento alterado na árvore, fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8
95	60	98	38	28	78	70	33

Exemplo 1: Alterar a prioridade do nó 6 de 66 para 98



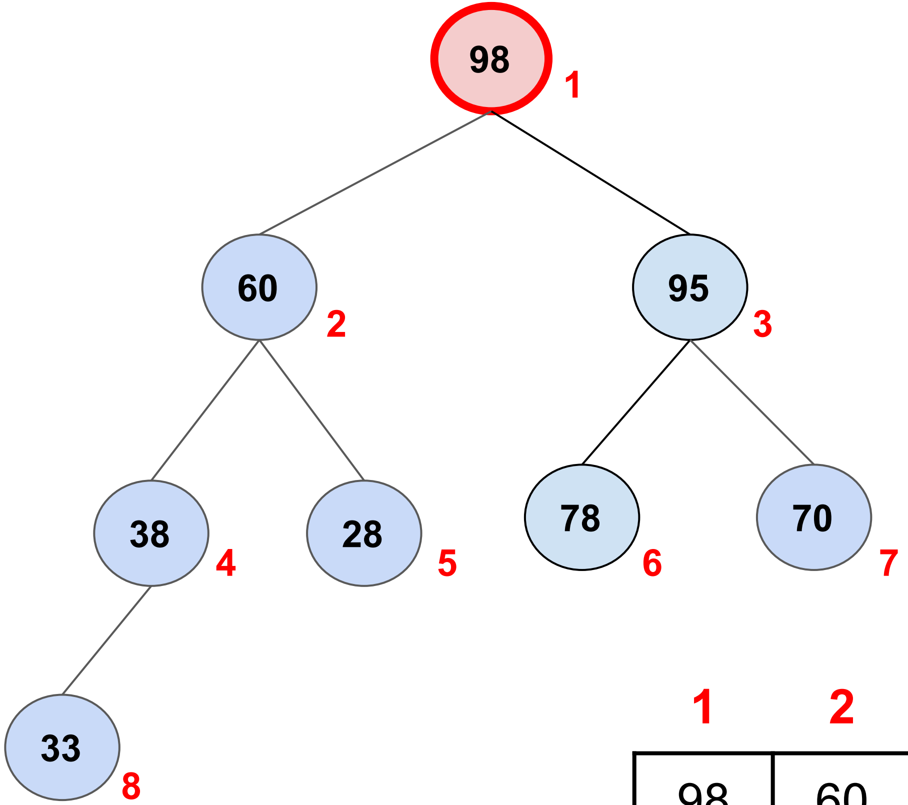
"Subir" elemento alterado na árvore, fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8
98	60	95	38	28	78	70	33

Função Subir

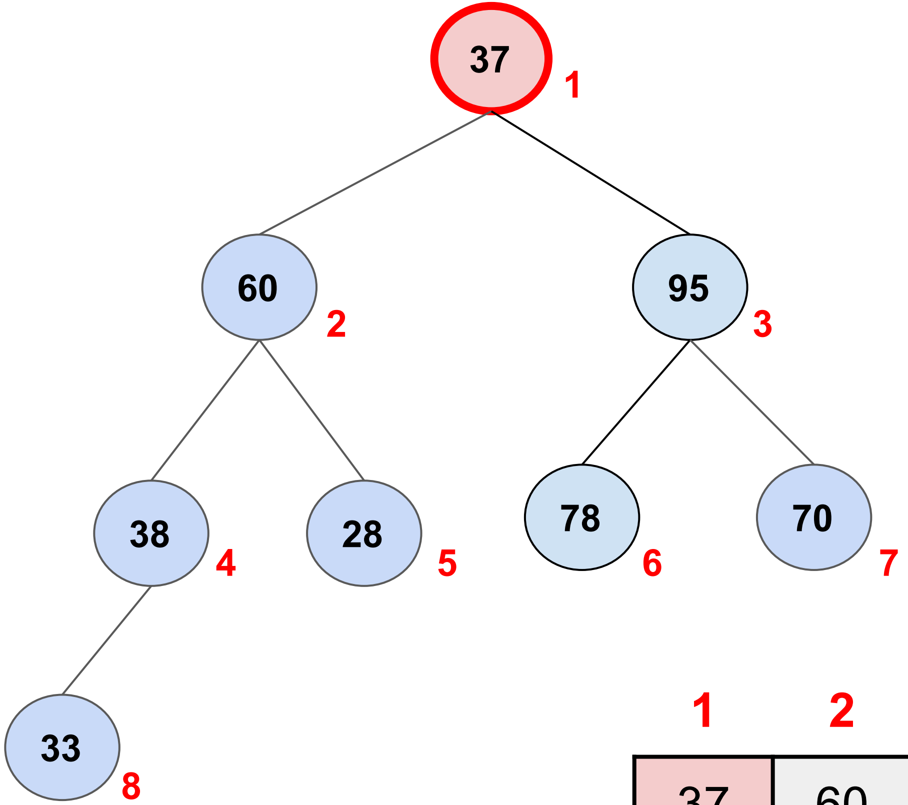
```
Dado o arranjo A e nó na posição i
procedimento subir(A,i)
p = Pai(i)
Se (p >= 1 ) Então
    Se A[i] > A[p] Então
        troca(A[i],A[p])
        subir(A,p)
    Fim-Se
Fim-Se
```

Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



1	2	3	4	5	6	7	8
98	60	95	38	28	78	70	33

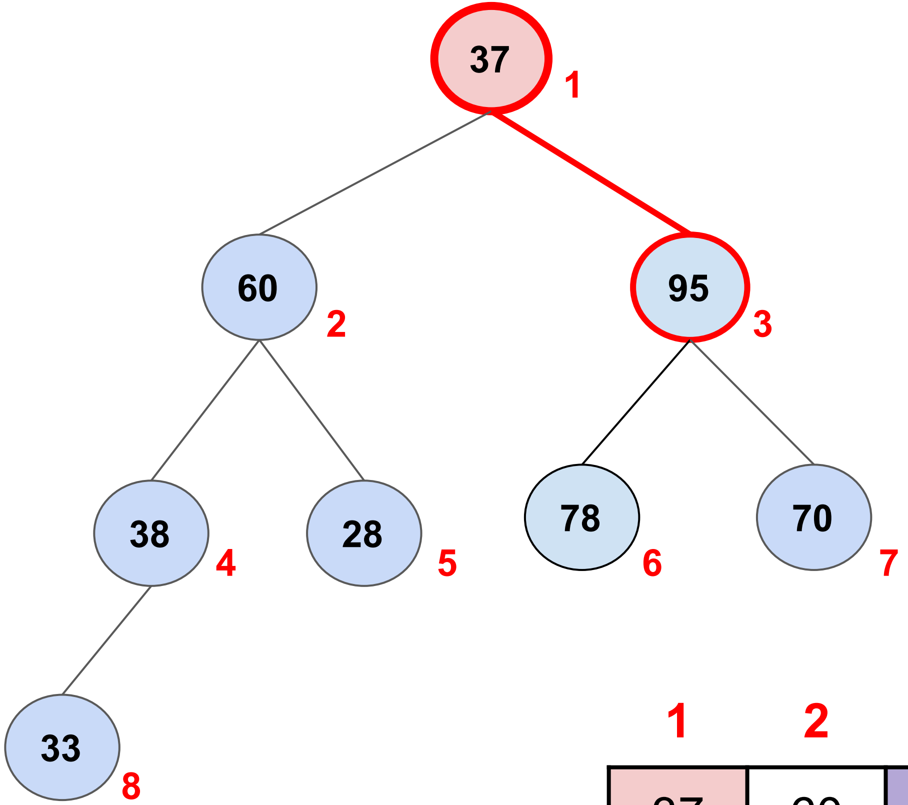
Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



"Descer" elemento alterado na árvore, fazendo trocas com o nó filho de maior prioridade, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
37	60	95	38	28	78	70	33

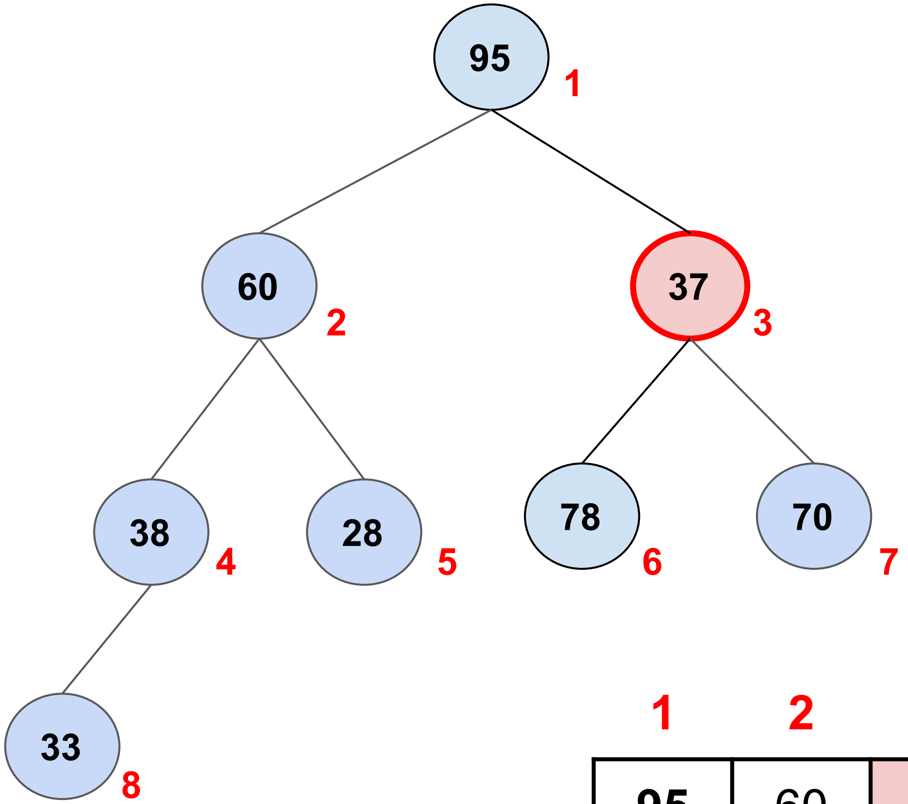
Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



"Descer" elemento alterado na árvore, fazendo trocas com o nó filho de maior prioridade, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
37	60	95	38	28	78	70	33

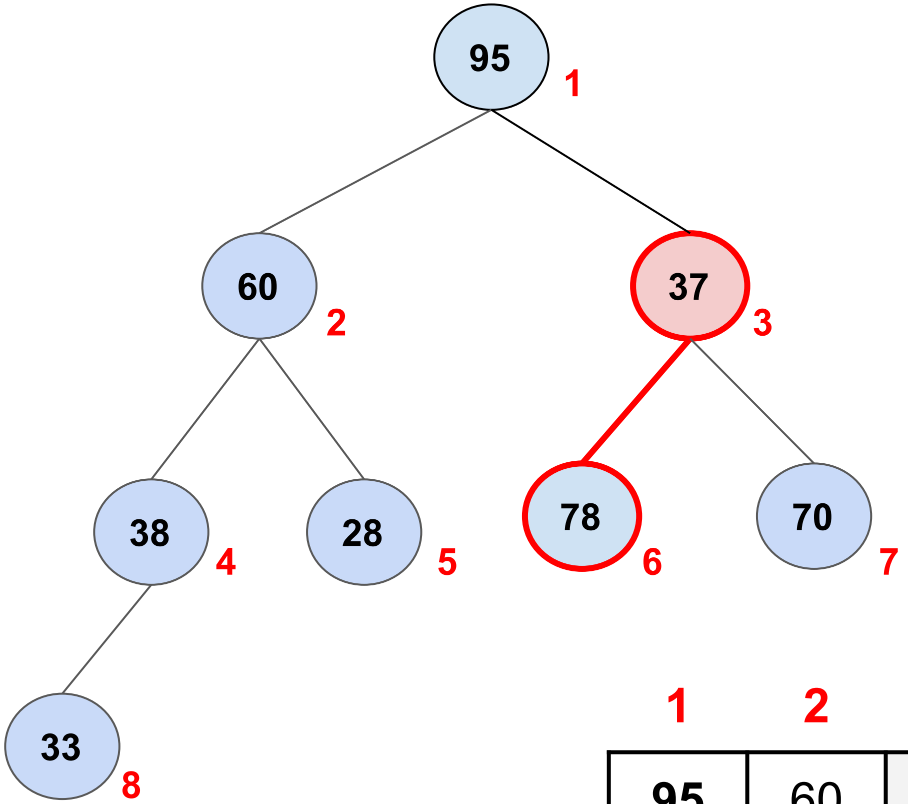
Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



"Descer" elemento alterado na árvore, fazendo trocas com o nó filho de maior prioridade, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
95	60	37	38	28	78	70	33

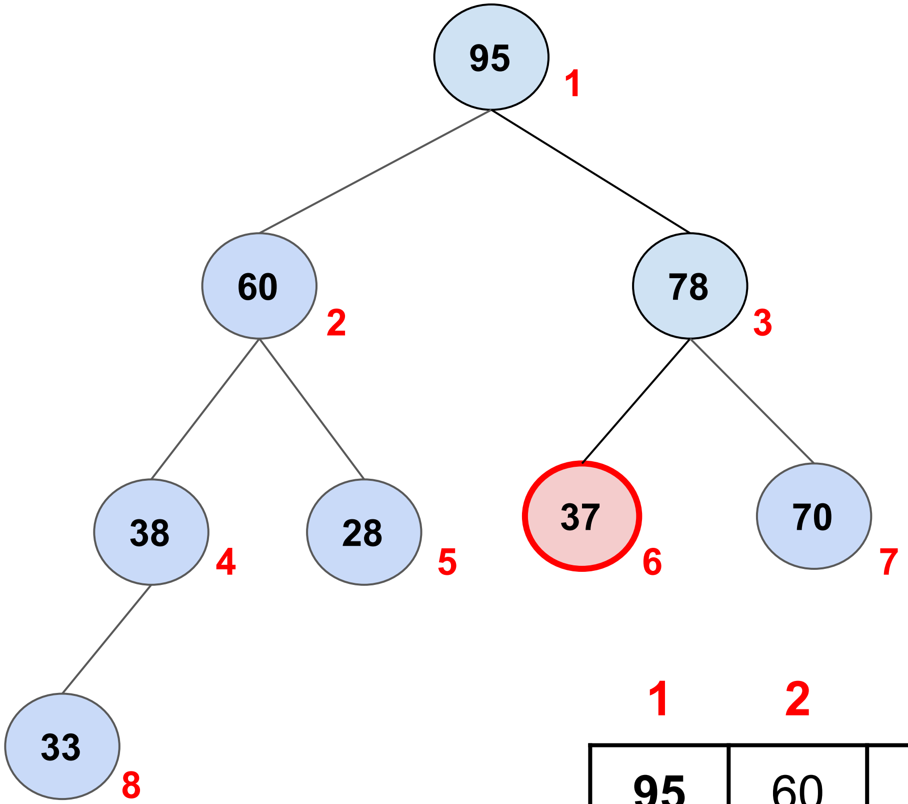
Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



"Descer" elemento alterado na árvore, fazendo trocas com o nó filho de maior prioridade, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
95	60	37	38	28	78	70	33

Exemplo 2: Alterar a prioridade do nó 1 de 98 para 37



"Descer" elemento alterado na árvore, fazendo trocas com o nó filho de maior prioridade, até que a árvore volte a ficar correta

1	2	3	4	5	6	7	8
95	60	**78**	38	28	37	70	33

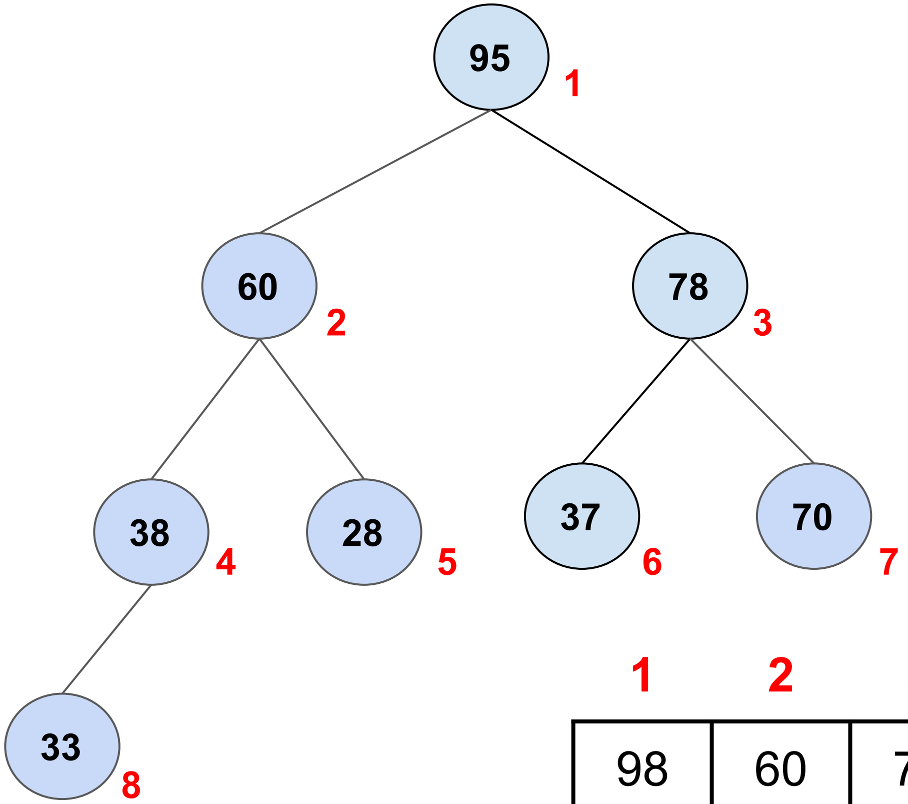
Função Descer

```
procedimento descer(A,i,n)
l = Esq(i)
r = Dir(i)
Se ( (l <= n) E (A[l] > A[i]) ) Então
    maior = l
Caso Contrário
    maior = i
Fim-Se
Se (r <= n) E (A[r] > A[maior] )
Então
     maior = r
Fim-Se
Se (i != maior) Então
     troca(A[i],A[maior])
     descer(A,maior,n)
Fim-Se
```

Operação: Inserção

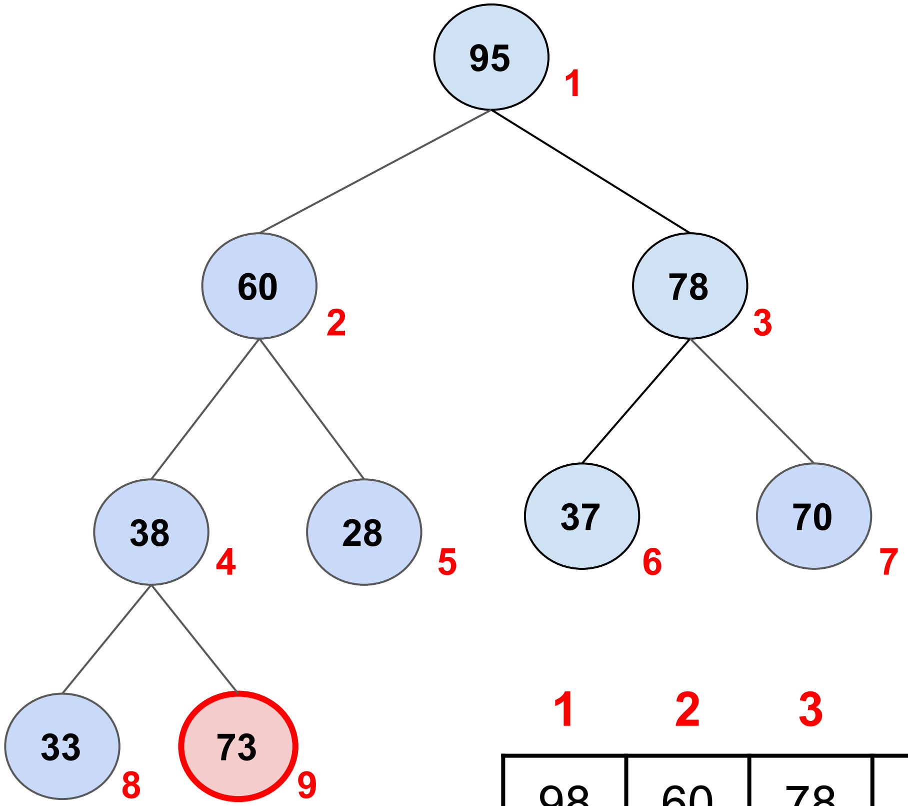
- Considerar uma heap com n elementos
- Inserir novo elemento na posição n + 1
- Assumir que esse elemento já existia e teve sua prioridade aumentada
- Chamar a função "subir" para colocar o elemento na posição correta na heap.

Exemplo 3: Inserir o elemento com prioridade 73 na heap



1	2	3	4	5	6	7	8
98	60	78	38	28	37	70	33

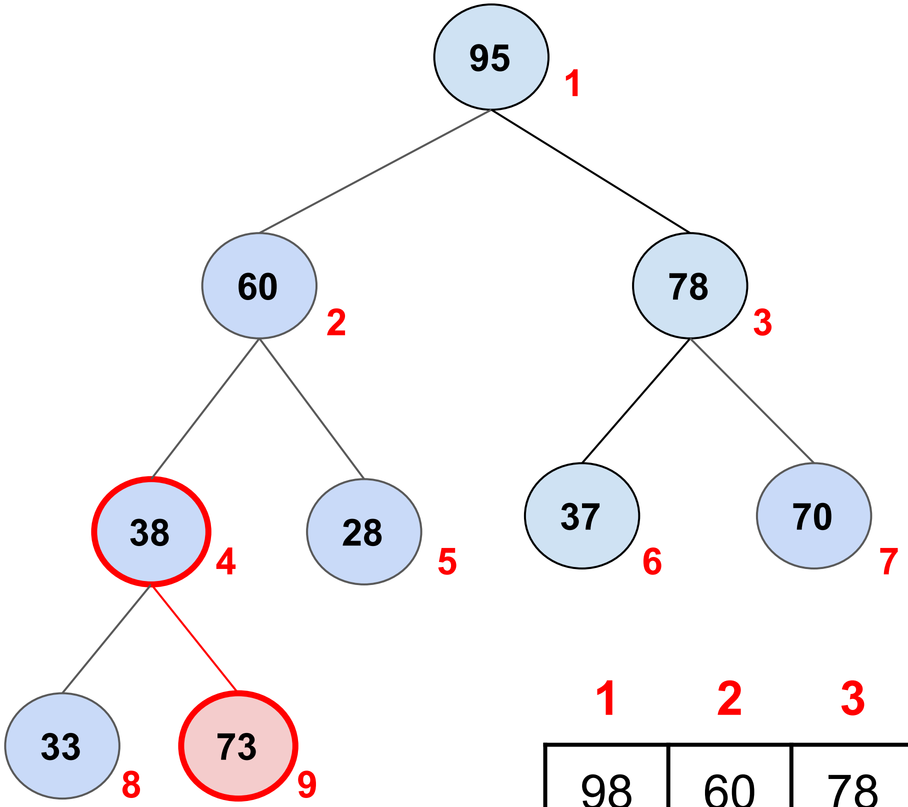
Exemplo 3: Inserir o elemento com prioridade 73 na heap



Colocar o elemento na posição n+1

1	2	3	4	5	6	7	8	9
98	60	78	38	28	37	70	33	73

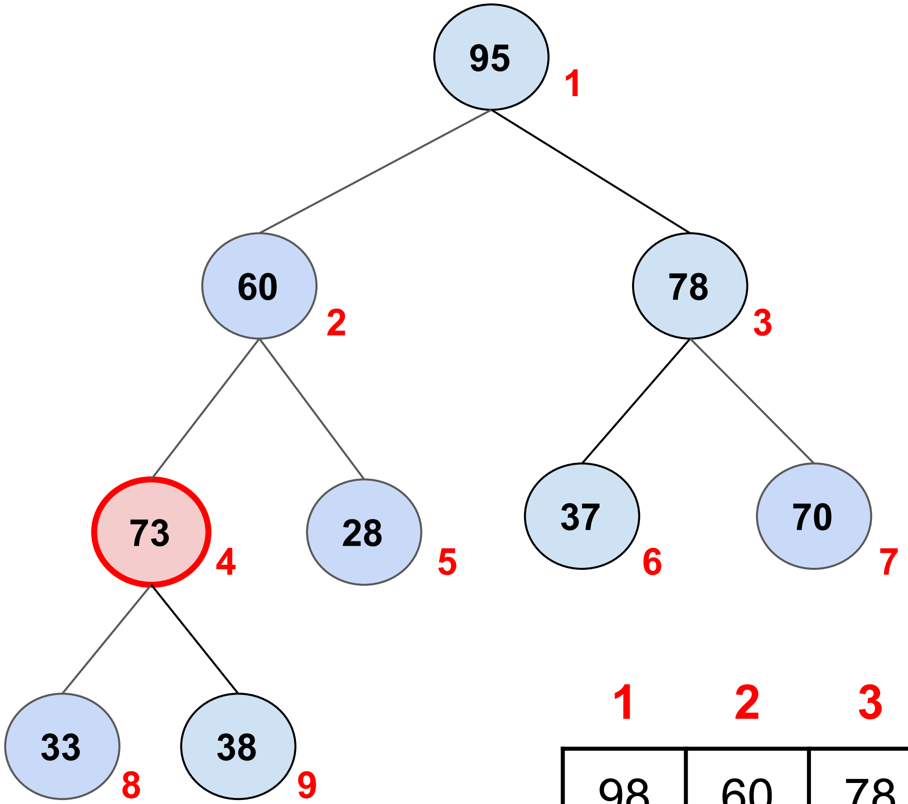
Exemplo 3: Inserir o elemento com prioridade 73 na heap



Subir o elemento,fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8	9
98	60	78	38	28	37	70	33	73

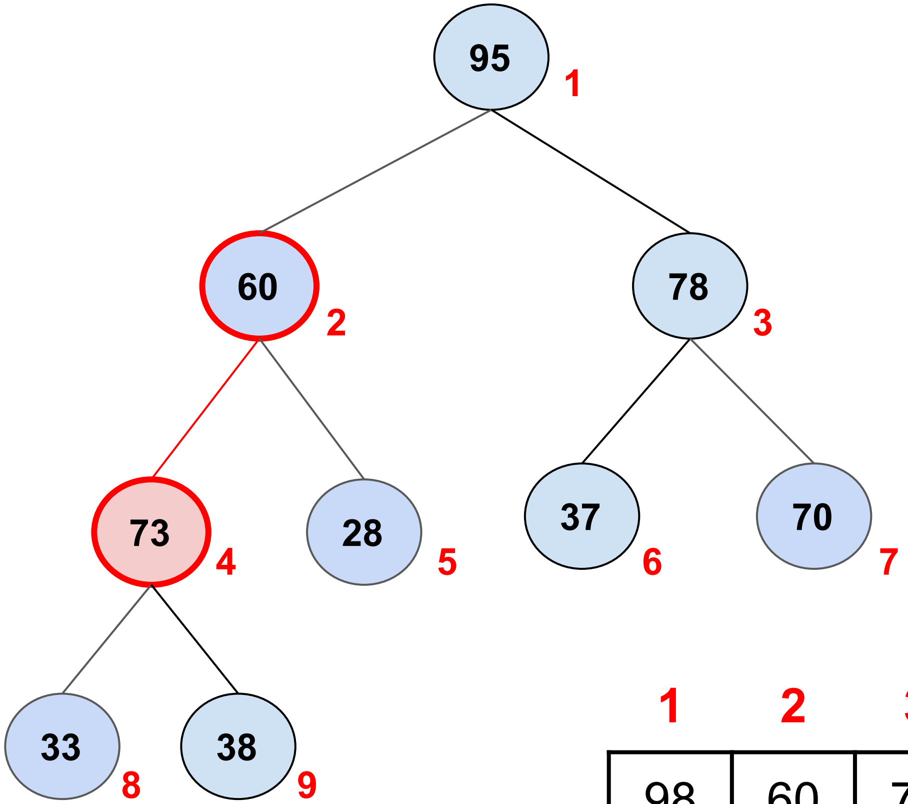
Exemplo 3: Inserir o elemento com prioridade 73 na heap



Subir o elemento,fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8	9
98	60	78	73	28	37	70	33	38

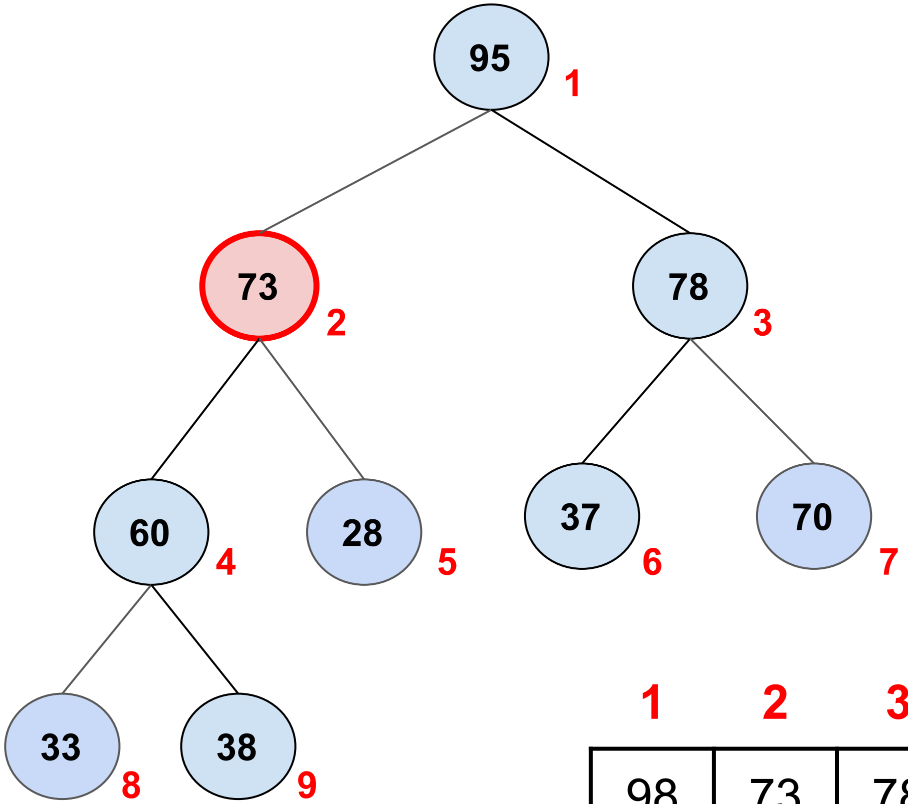
Exemplo 3: Inserir o elemento com prioridade 73 na heap



Subir o elemento,fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8	9
98	60	78	73	28	37	70	33	38

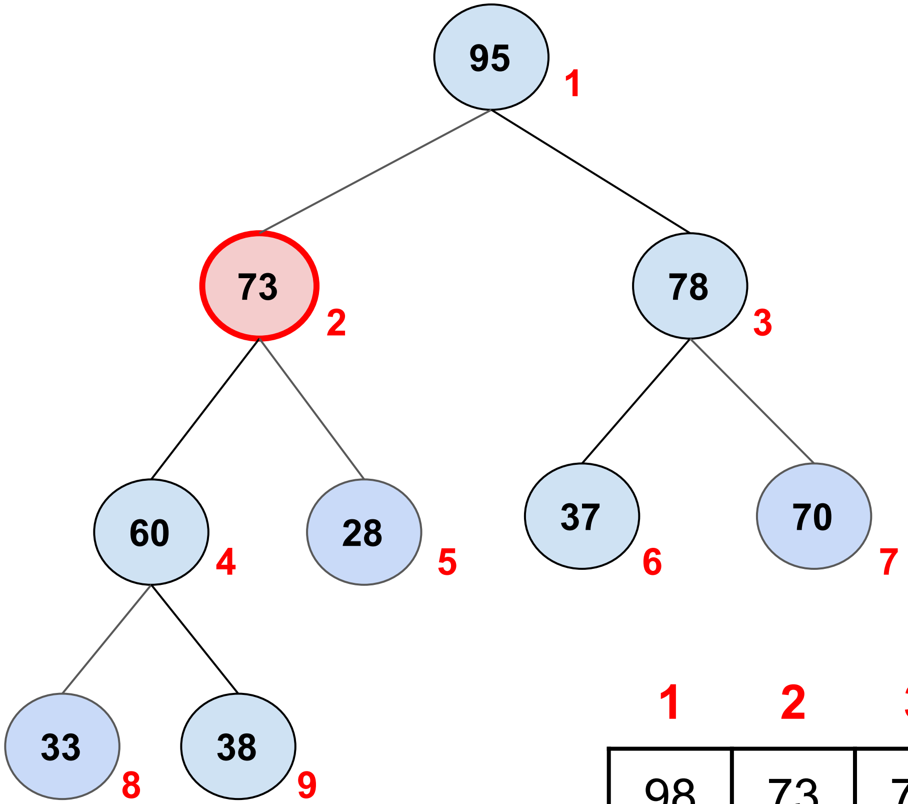
Exemplo 3: Inserir o elemento com prioridade 73 na heap



Subir o elemento,fazendo trocas com o nó pai, até que a árvore volte a ficar correta.

1	2	3	4	5	6	7	8	9
98	73	78	60	28	37	70	33	38

Exemplo 3: Inserir o elemento com prioridade 73 na heap



Elemento estaciona na posição 2 pois sua prioridade é inferior ao seu nó pai.

1	2	3	4	5	6	7	8	9
98	73	78	60	28	37	70	33	38

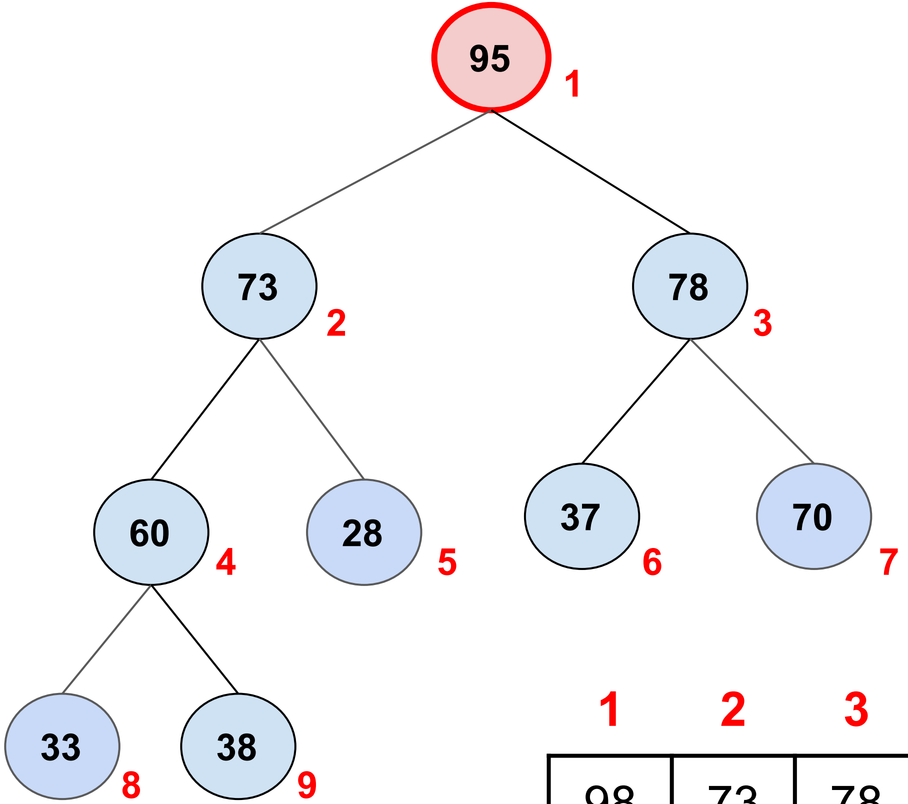
Função Insere

```
procedimento inserir(A,chave, n)
   1.  n    = n + 1
   2.  A[n] = chave
   3.  subir(A,n)
```

Operação: Remoção do elemento de maior prioridade

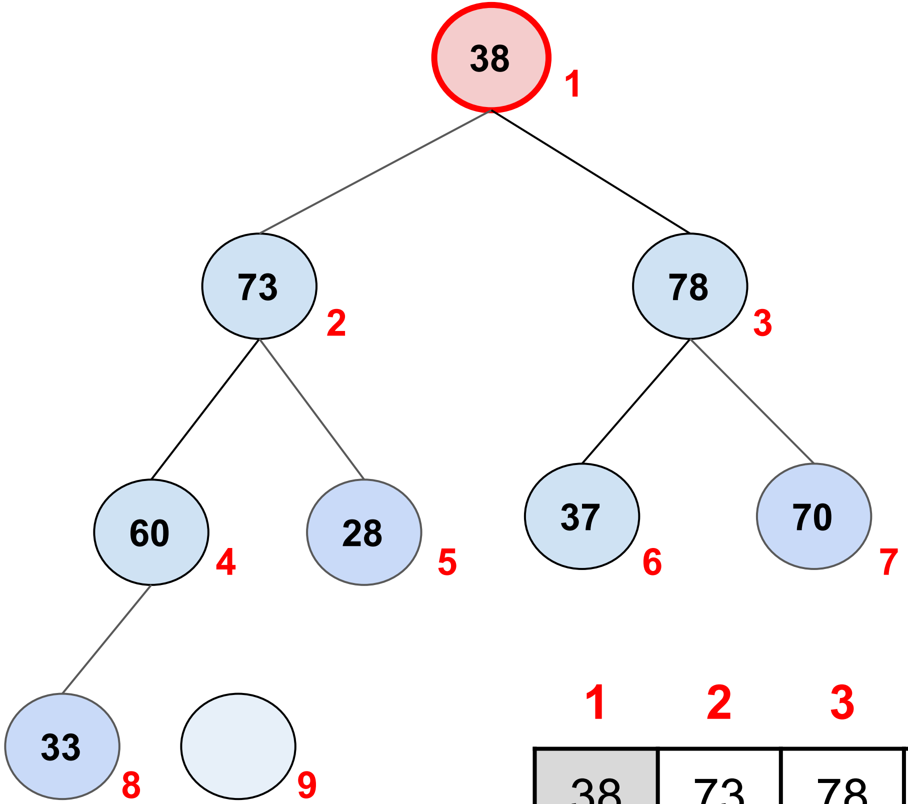
- Remover o primeiro elemento da heap
- Preencher o espaço vazio deixado por ele com o último elemento da heap
- Executar o algoritmo de descida na árvore para corrigir a prioridade desse elemento

Exemplo 4: Remover o elemento de maior prioridade



1	2	3	4	5	6	7	8	9
98	73	78	60	28	37	70	33	38

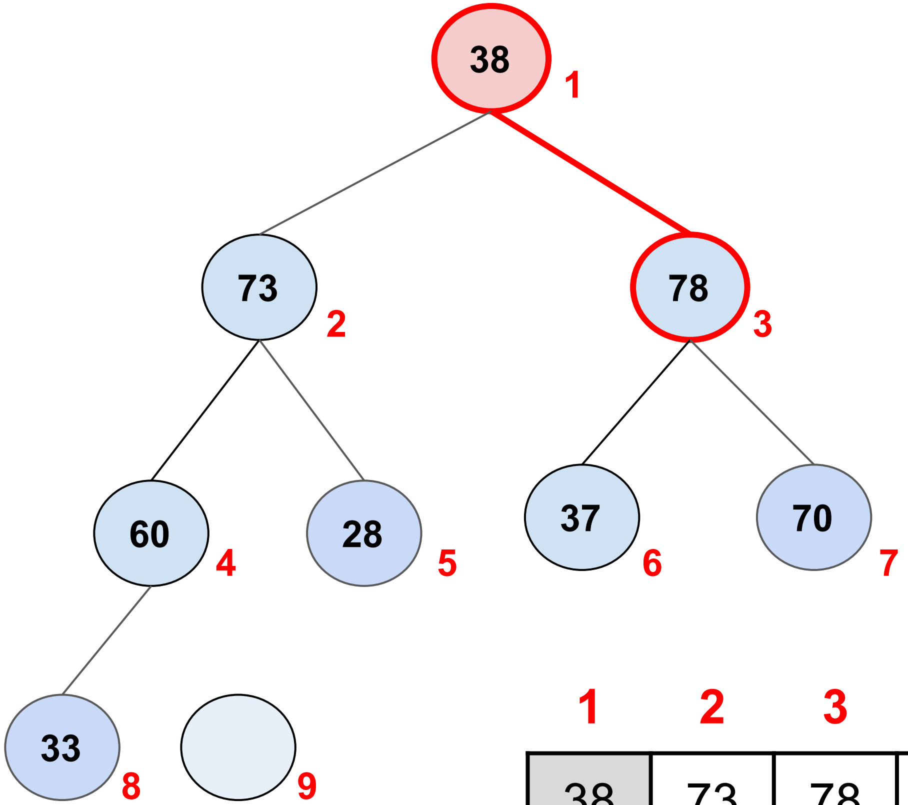
Exemplo 4: Remover o elemento de maior prioridade



Remover o primeiro elemento da heap. Preencher o espaço vazio deixado por ele com o último elemento da heap

1	2	3	4	5	6	7	8	9
38	73	78	60	28	37	70	33	

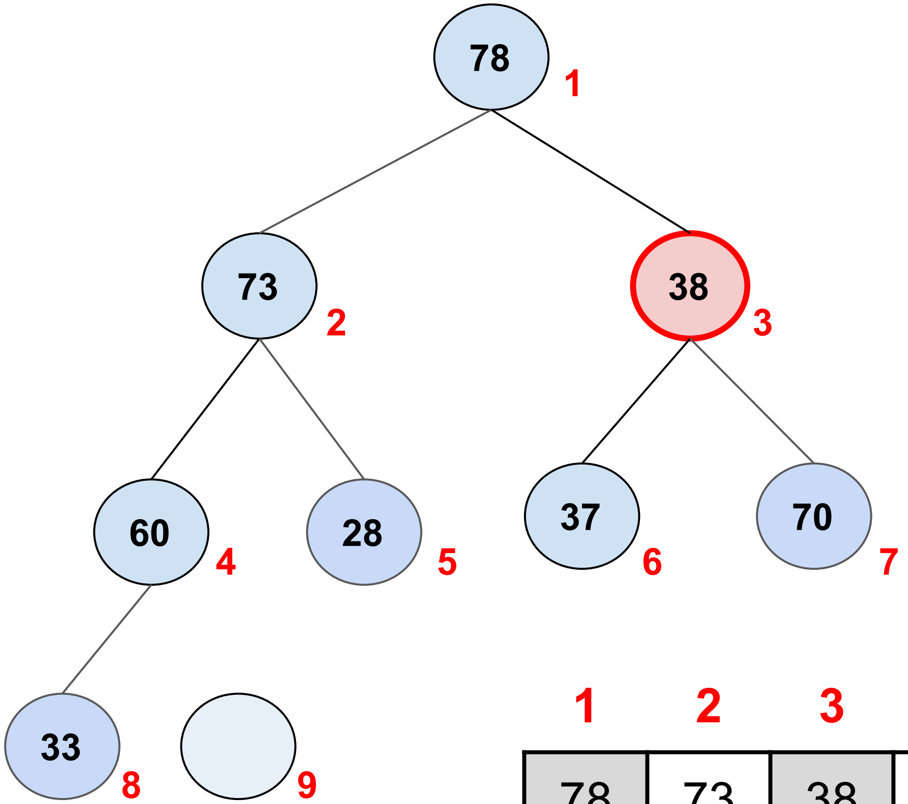
Exemplo 4: Remover o elemento de maior prioridade



Descer na árvore para corrigir a prioridade desse elemento

1	2	3	4	5	6	7	8	9
38	73	78	60	28	37	70	33	

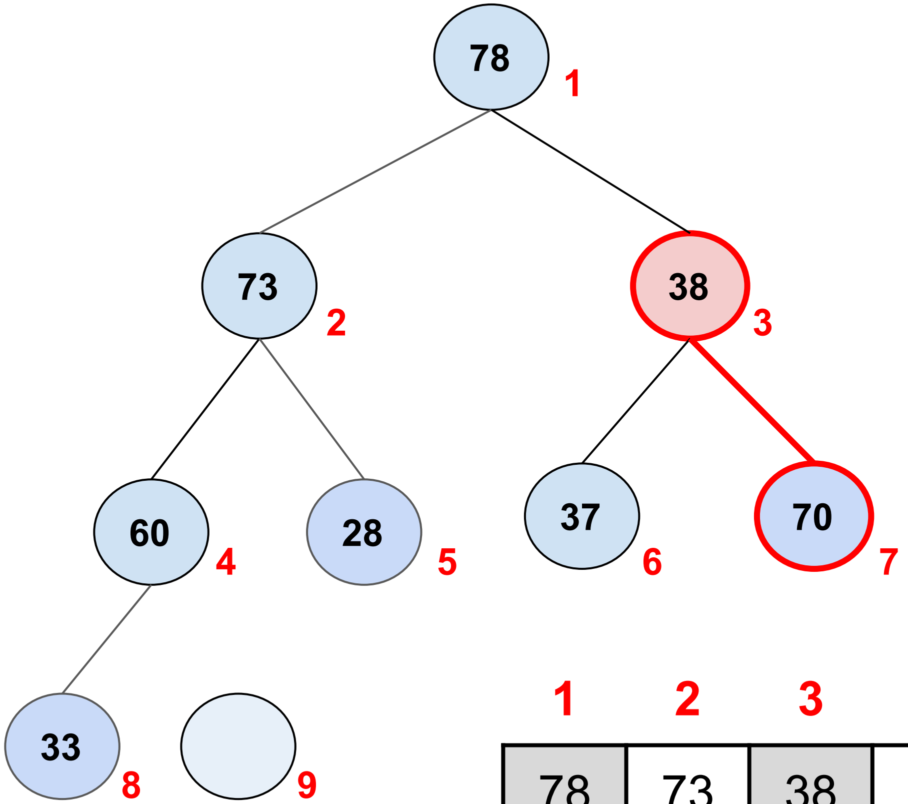
Exemplo 4: Remover o elemento de maior prioridade



Descer na árvore para corrigir a prioridade desse elemento

1	2	3	4	5	6	7	8	9
78	73	38	60	28	37	70	33	

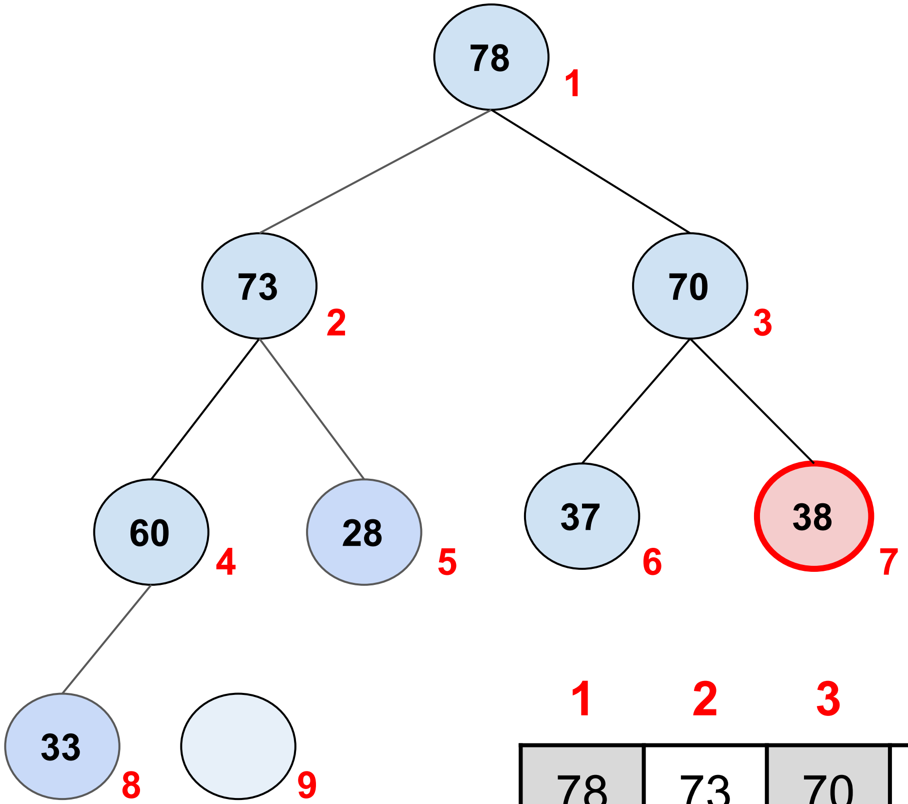
Exemplo 4: Remover o elemento de maior prioridade



Descer na árvore para corrigir a prioridade desse elemento

1	2	3	4	5	6	7	8	9
78	73	38	60	28	37	70	33	

Exemplo 4: Remover o elemento de maior prioridade



Descer na árvore para corrigir a prioridade desse elemento

1	2	3	4	5	6	7	8	9
78	73	70	60	28	37	38	33	

Função Remover

```
procedimento Remover(A,n)
Se (n < 1) então
      error "heap underflow"
Fim-Se
max  = A[1]
A[1] = A[n]
n    = n - 1
descer(A,1,n)
return max
```

Visualização:

Visualização: http://btv.melezinek.cz/binary-heap.html

Nova Operação

Fusão de Listas de Prioridades

Filas de Prioridades Avançadas

- Implementação eficiente através de Heap.
- Dada uma lista com n elementos:
  - **Seleção** em tempo **O(1)**
  - **Inserção** e **remoção** em em tempo **O(log n)**
  - **Construção** em tempo **O(n)**.
- **Aplicações avançadas necessitam unir** filas de prioridades de forma eficiente. **Com fazer isso?**
  - **Usar generalizações do heaps que permitem complexidade O(log n)**

Heap Esquerdistas

Heap Binomial

Heap Esquerdistas

Heap Esquerdistas

Relembrando:

Um heap corresponde um conjunto de valores numéricos, denominados prioridade, que são associados aos nós da estrutura T, satisfazendo duas condições:

(1) Se v é o pai de w em T, a prioridade de v é maior ou igual à de w.
(2) T é uma árvore binária quase completa, em que todos os nós do seu último nível se encontra mais à esquerda possível.

Heaps esquerdistas serão definidos mediante uma modificação desta última condição.

Árvores Esquerdistas

Algumas definições:

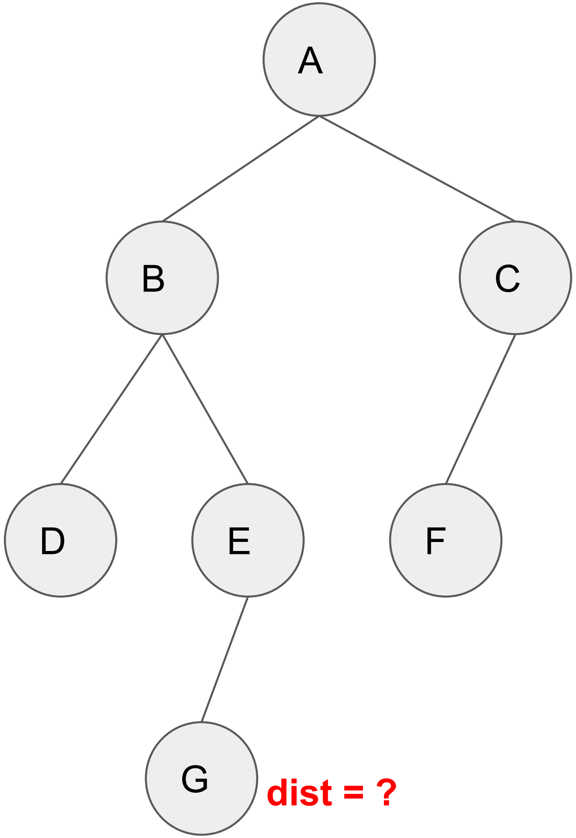
- Cada nó x da árvore esquerdista tem quatro campos:
  - *chave*: prioridade do nó
  - *esq*(x): filho esquerdo de x;
  - *dir*(x) : filho direito de x;
  - *dist*(x): menor comprimento do nó x até o nó folha

Chave	
dist	
esq	dir

```
procedimento dist(x)
    Se x == NULL Então
          retorna 0
    Fim-Se
    retorna 1 + min(dist(Esq(i)),dist(Dir(i))
```

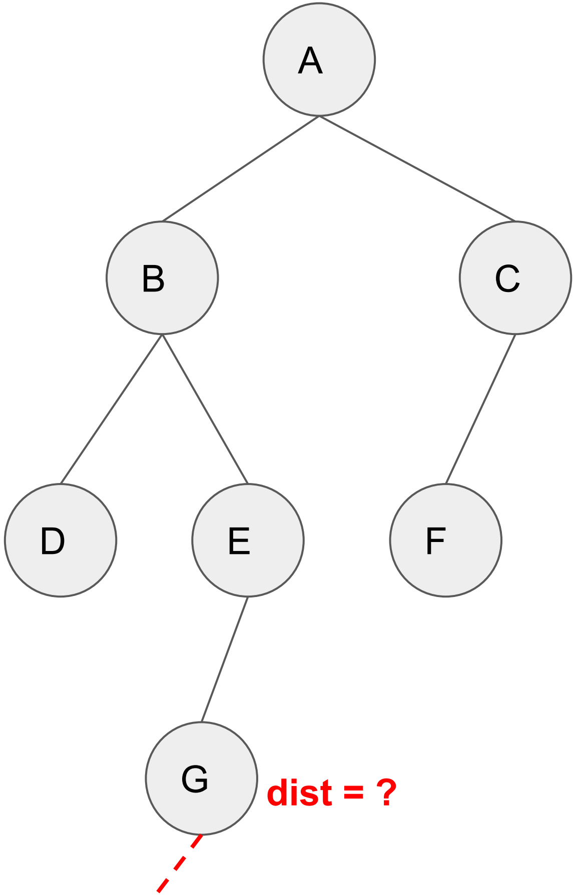
Exemplificando caminho de menor comprimento...

```
procedimento dist(x)
1.   Se x == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 +
     min(dist(Esq(i)),dist(Dir(i))
```



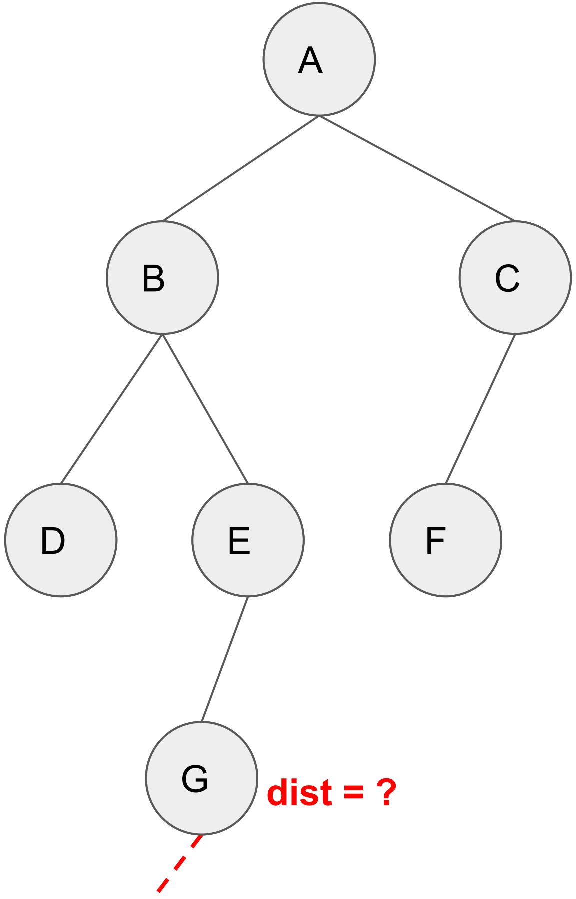
Exemplificando caminho de menor comprimento...

```
procedimento dist(G)
1.   Se G == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 +
     min(dist(Esq(G)),dist(Dir(G))
```



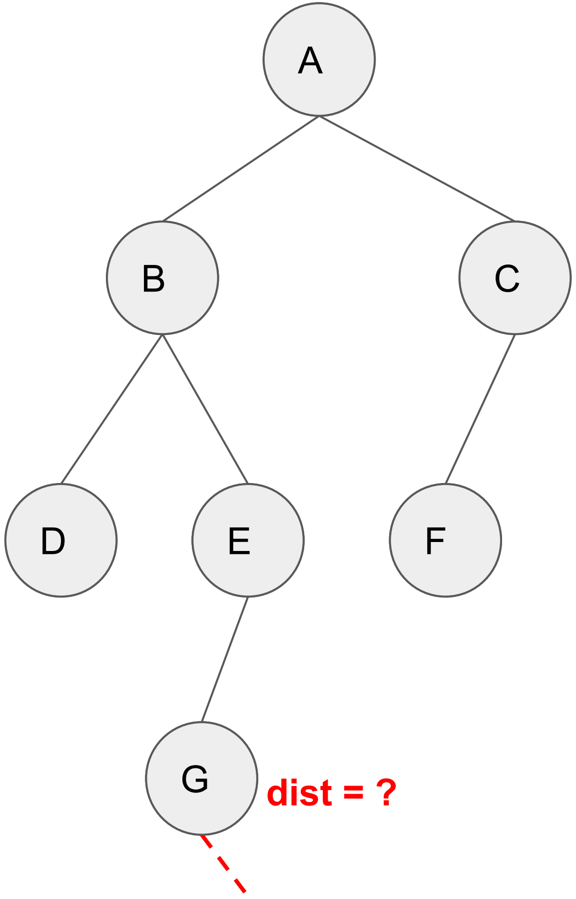
Exemplificando caminho de menor comprimento...

```
procedimento dist(G)
    Se G == NULL Então
          retorna 0
    Fim-Se
    retorna 1 + min(0,dist(Dir(G)))
```



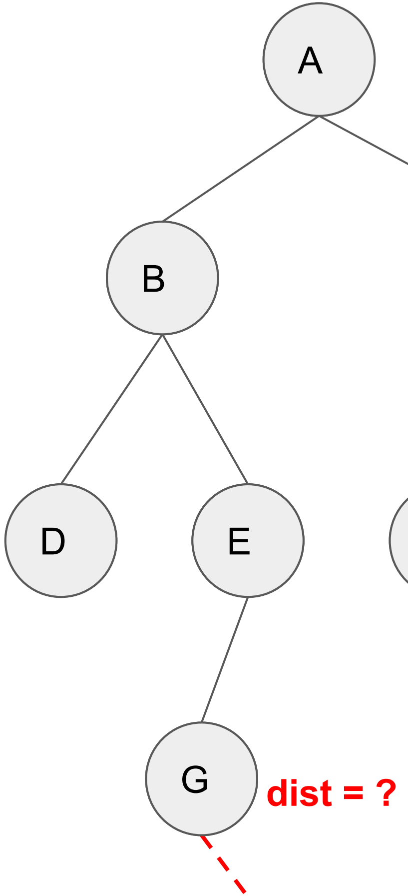
Exemplificando caminho de menor comprimento...

```
procedimento dist(G)
    Se G == NULL Então
          retorna 0
    Fim-Se
    retorna 1 + min(0,dist(dir(G)))
```



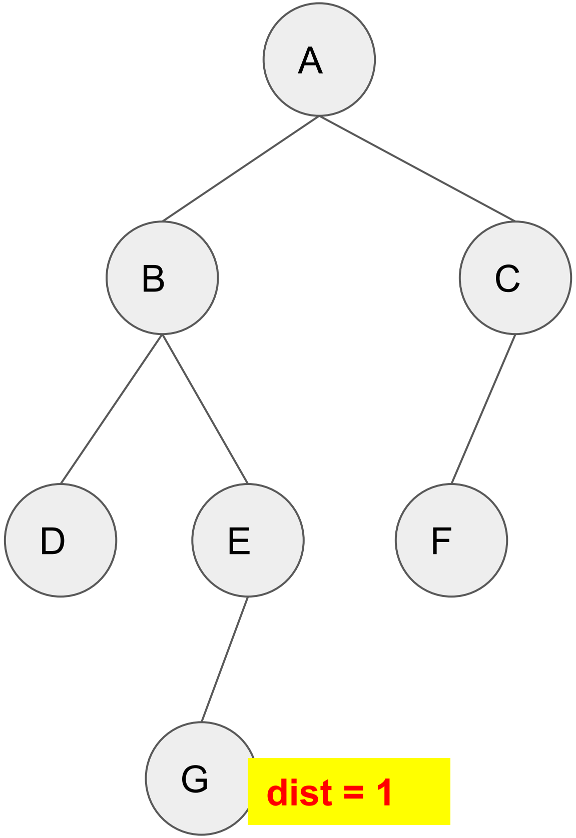
Exemplificando caminho de menor comprimento...

```
procedimento dist(G)
1.   Se G == NULL Então
2.          retorna 0
3.   Fim-Se
4.   retorna 1 + min(0,0)
```



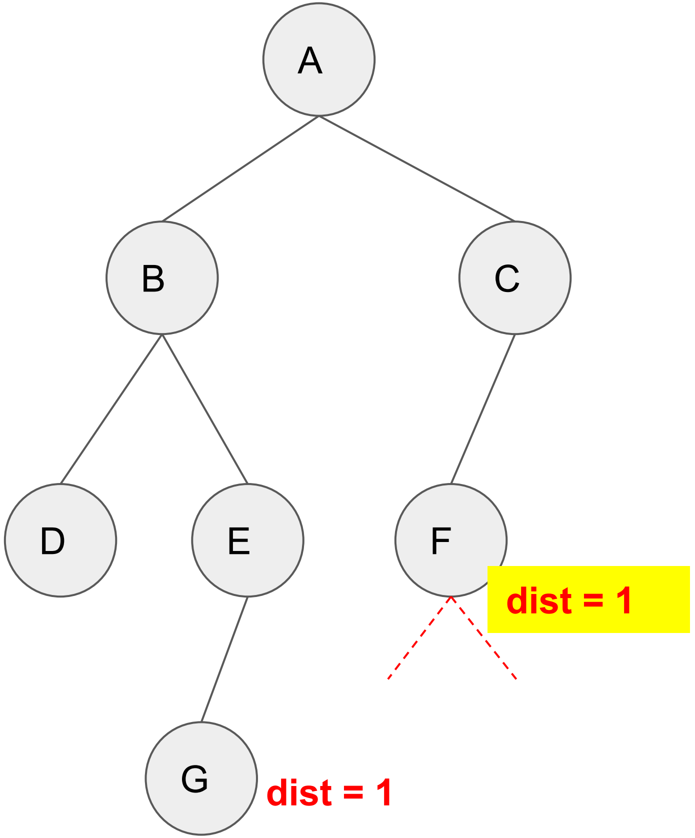
Exemplificando caminho de menor comprimento...

```
procedimento dist(G)
    Se G == NULL Então
            retorna 0
    Fim-Se
    retorna 1 + min(0,0)
```



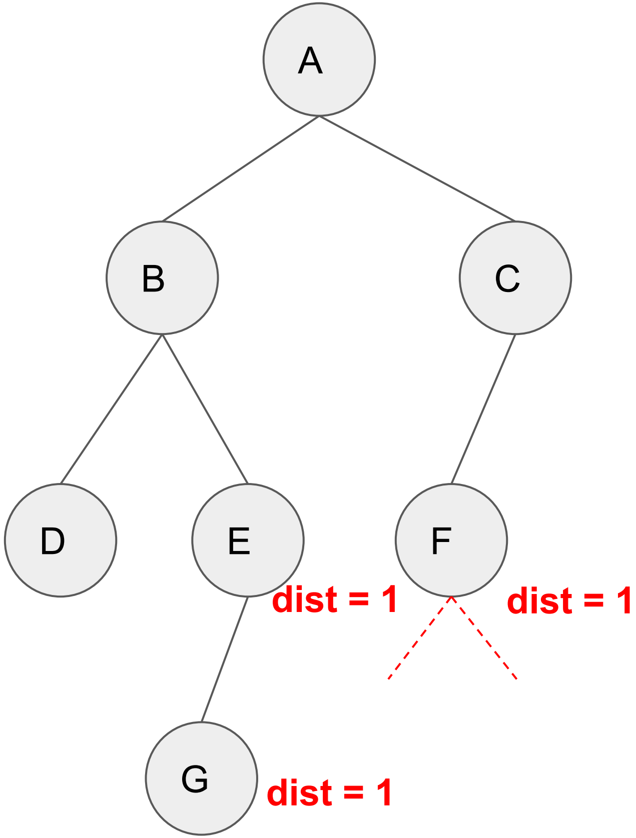
Exemplificando caminho de menor comprimento...

```
procedimento dist(F)
    Se x == NULL Então
            retorna 0
    Fim-Se
    retorna 1 + min(0,0)
```



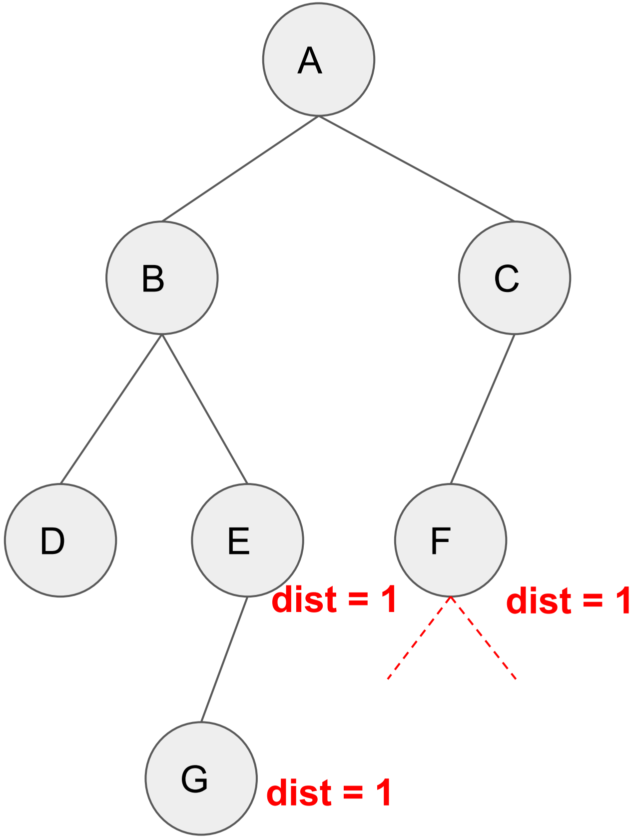
Exemplificando caminho de menor comprimento...

```
procedimento dist(E)
1.   Se E == NULL Então
2.           retorna 0
3.   Fim-Se
4.   retorna 1 +
     min(dist(Esq(E)),dist(dir(E)))
```



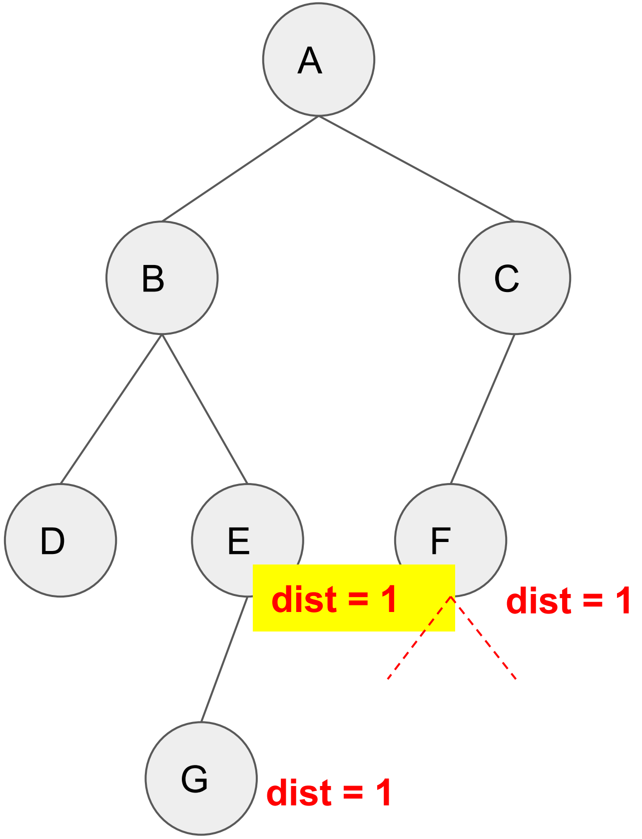
Exemplificando caminho de menor comprimento...

```
procedimento dist(E)
    Se E == NULL Então
        retorna 0
    Fim-Se
    retorna 1 + min(1,0)
```



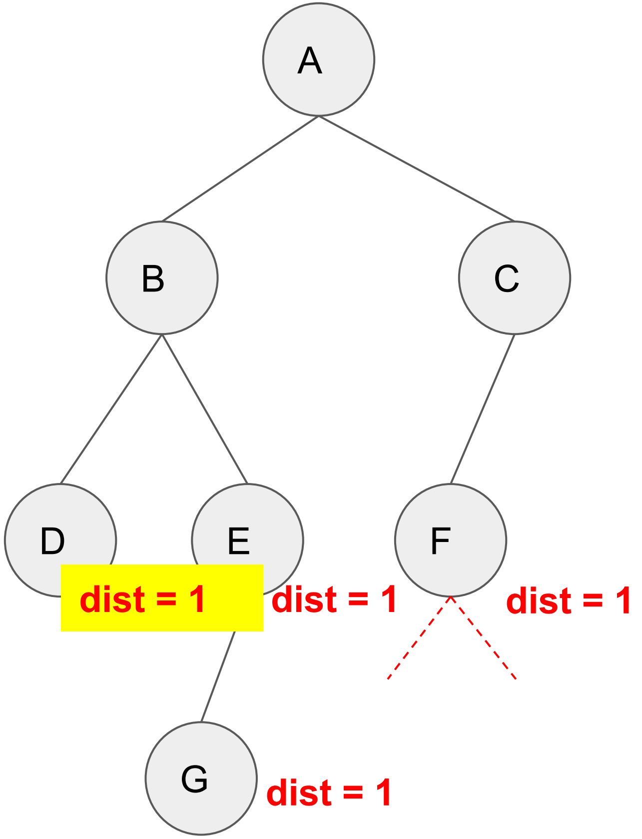
Exemplificando caminho de menor comprimento...

```
procedimento dist(E)
1.   Se E == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 + 0
```

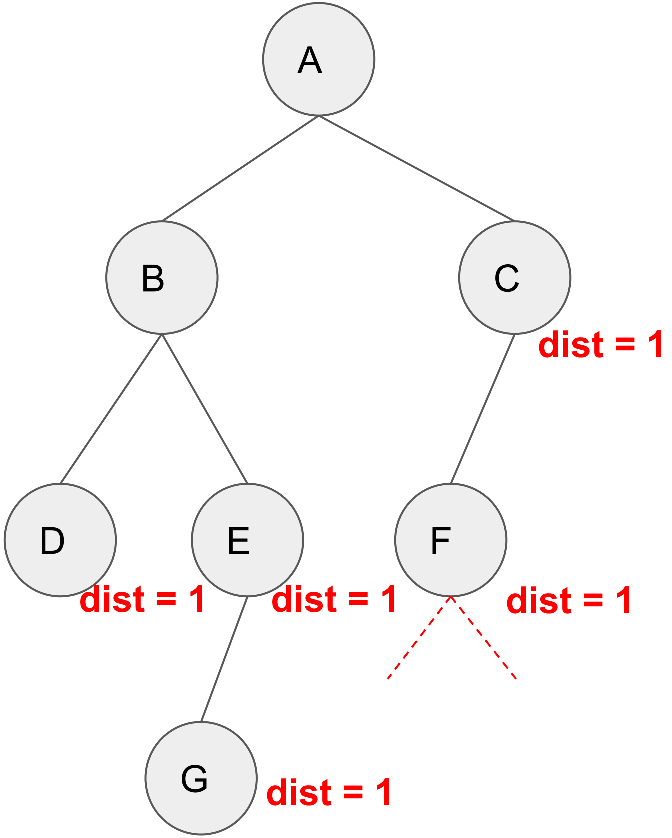


Exemplificando caminho de menor comprimento...

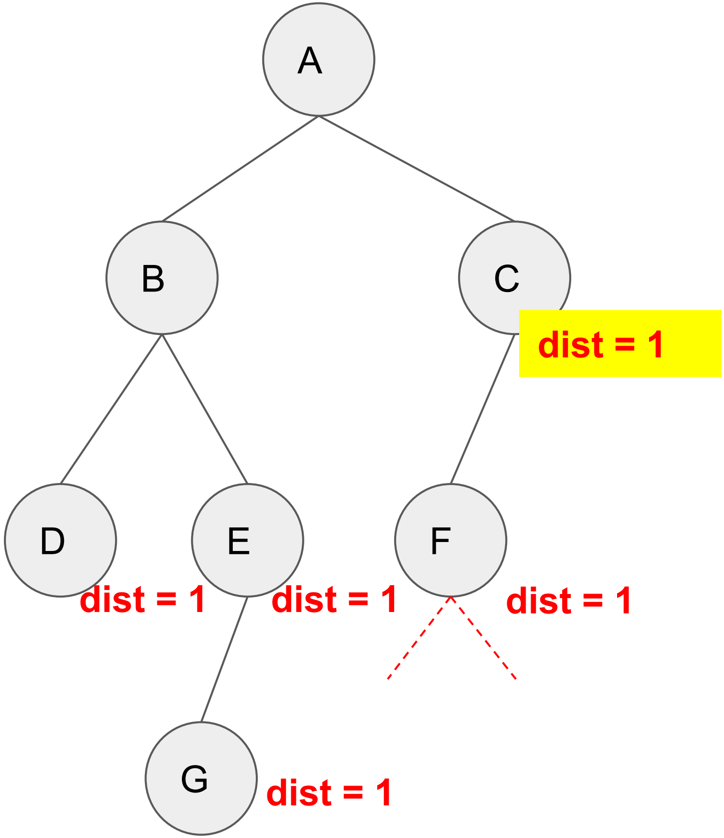
```
procedimento dist(D)
1.   Se E == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 + min(0,0)
```



```
procedimento dist(C)
1.   Se C == NULL Então
2.           retorna 0
3.   Fim-Se
4.   retorna 1 + min(1,0)
```

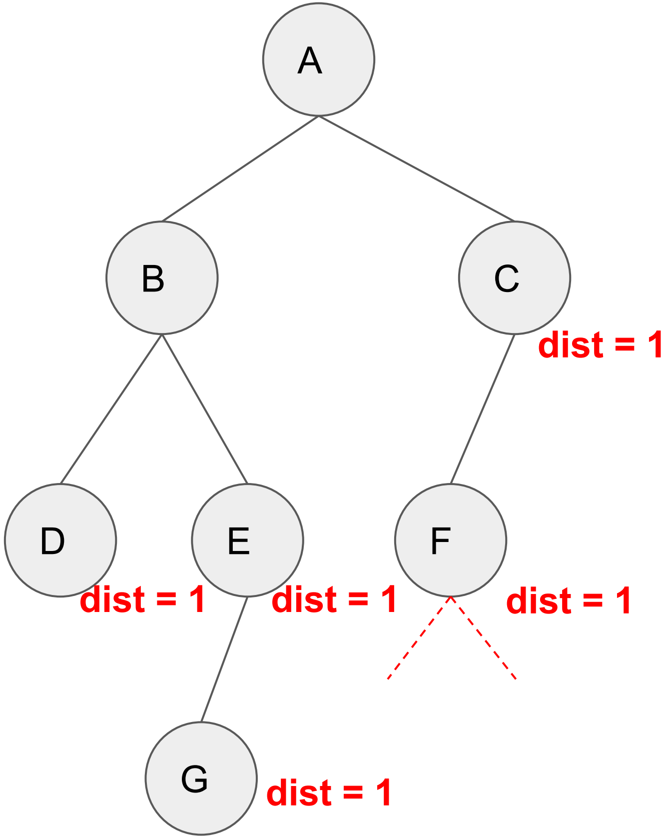


```
procedimento dist(C)
1.  Se C == NULL Então
2.        retorna 0
3.  Fim-Se
4.  retorna 1 + 0
```



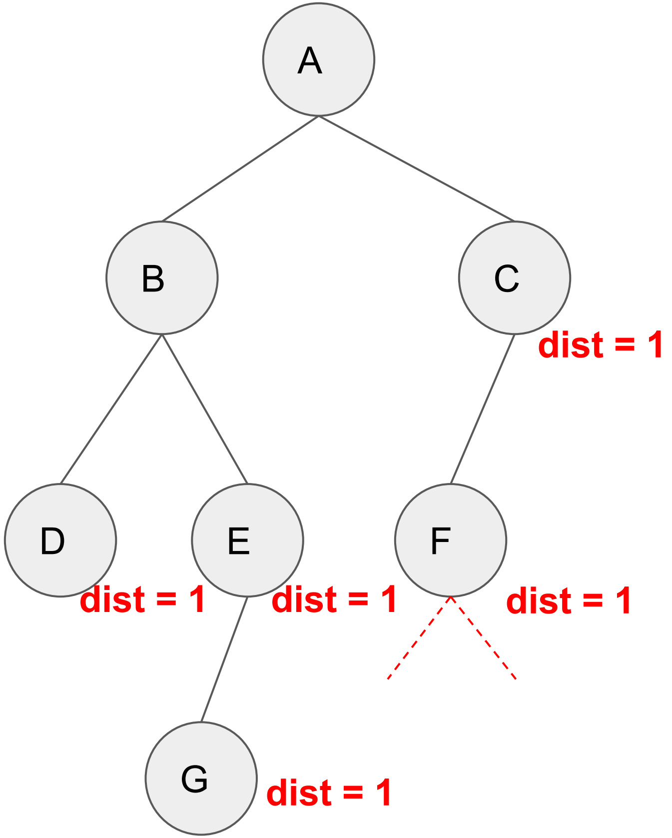
Exemplificando caminho de menor comprimento...

```
procedimento dist(B)
    Se B == NULL Então
            retorna 0
    Fim-Se
    retorna 1 +
    min(dist(D),dist(E))
```



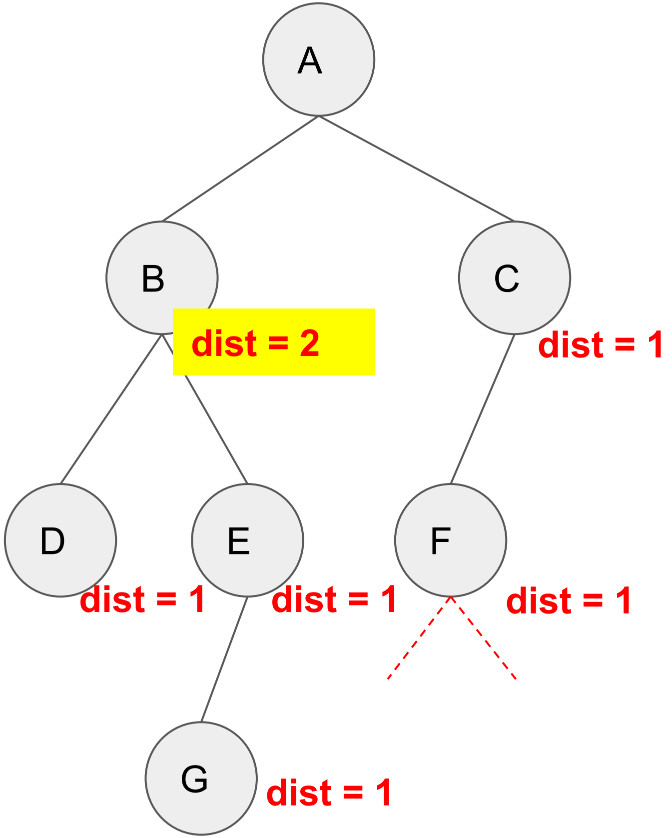
Exemplificando caminho de menor comprimento...

```
procedimento dist(B)
1.   Se B == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 + min(1,1)
```



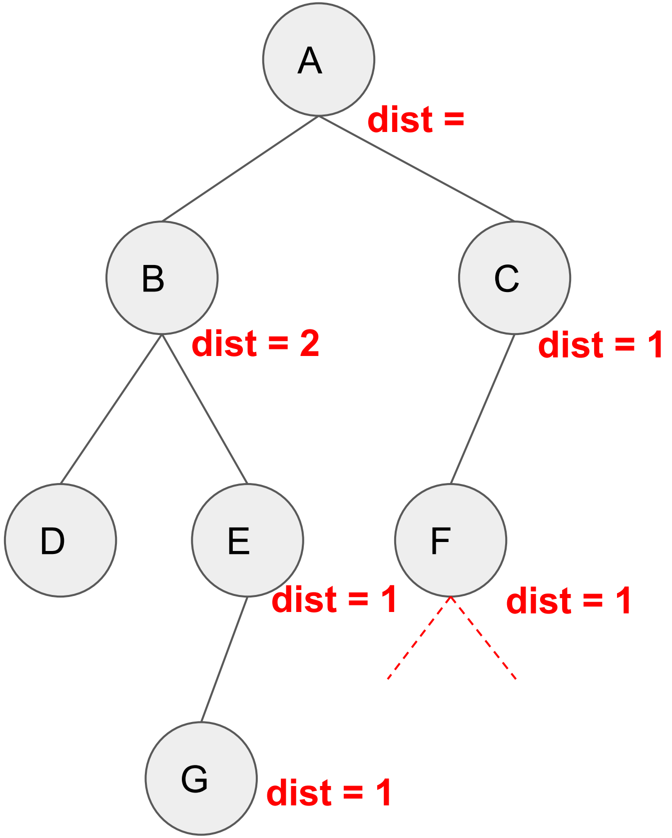
Exemplificando caminho de menor comprimento...

```
procedimento dist(B)
    Se B == NULL Então
            retorna 0
    Fim-Se
    retorna 1 + 1
```



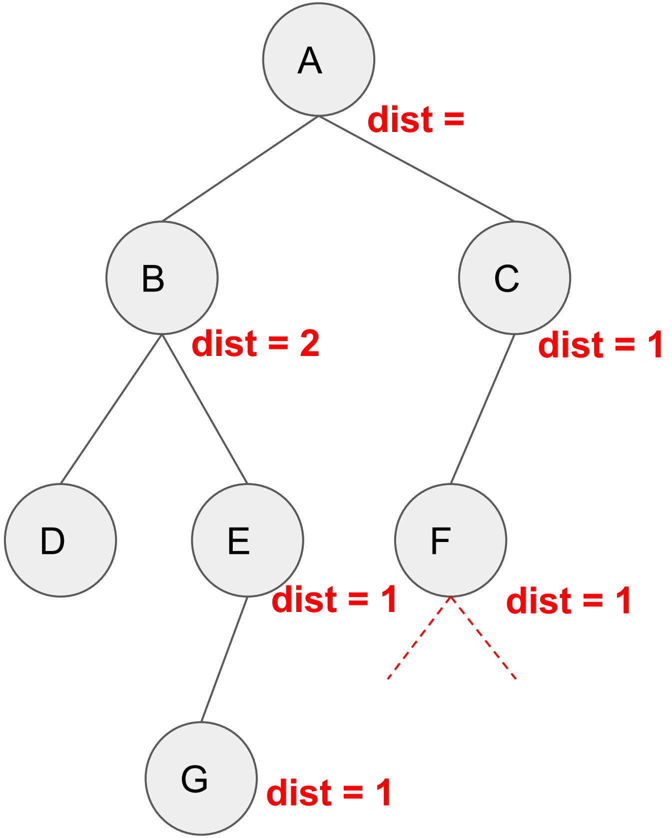
Exemplificando caminho de menor comprimento...

```
procedimento dist(A)
1.  Se A == NULL Então
2.          retorna 0
3.  Fim-Se
4.  retorna 1 + min(dist(Esq(A)),
    dist(Dir(A)))
```



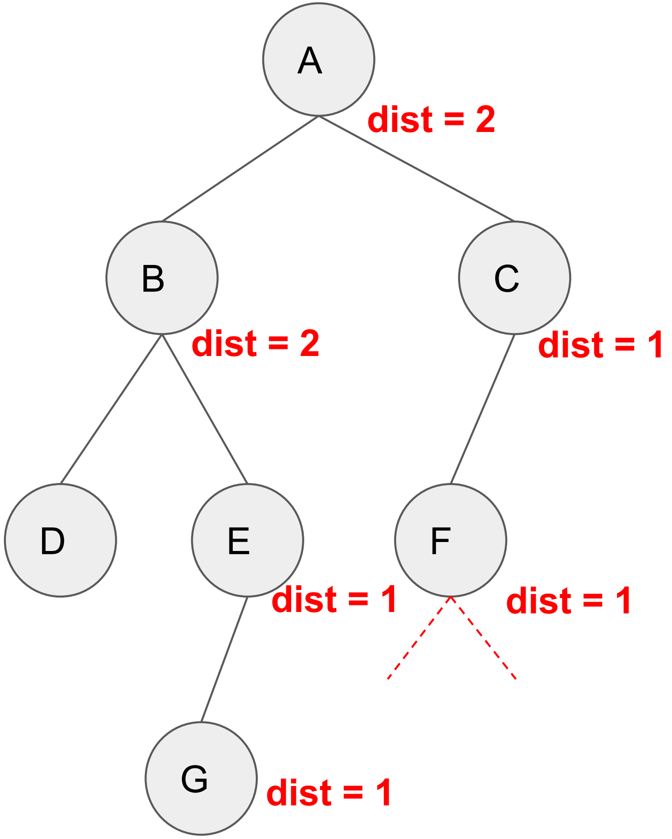
Exemplificando caminho de menor comprimento...

```
procedimento dist(A)
   Se A == NULL Então
         retorna 0
   Fim-Se
   retorna 1 + min(dist(B),
   dist(C))
```



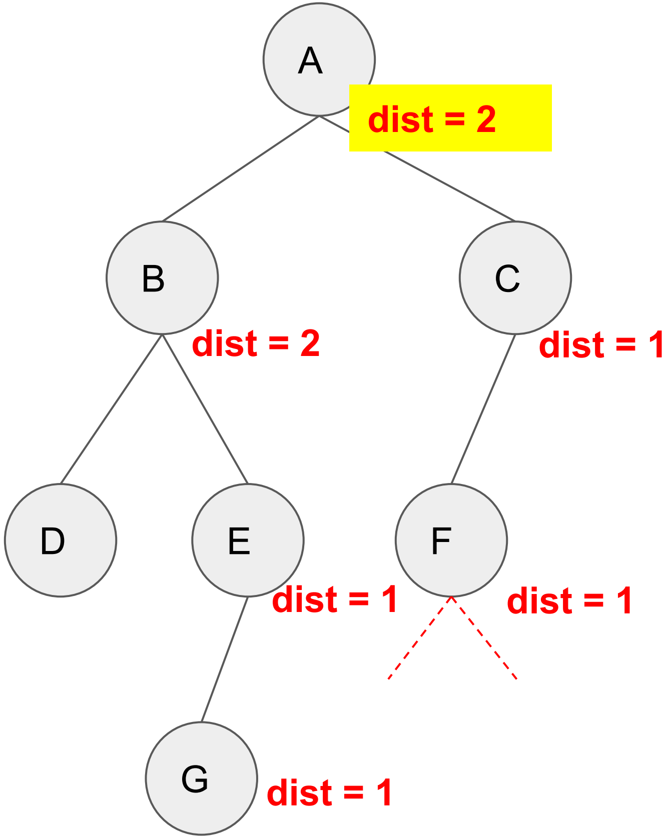
Exemplificando caminho de menor comprimento...

```
procedimento dist(A)
1.   Se A == NULL Então
2.        retorna 0
3.   Fim-Se
4.   retorna 1 + min(2, 1)
```



Exemplificando caminho de menor comprimento...

```
procedimento dist(A)
1.   Se A == NULL Então
2.           retorna 0
3.   Fim-Se
4.   retorna 1 + 1
```

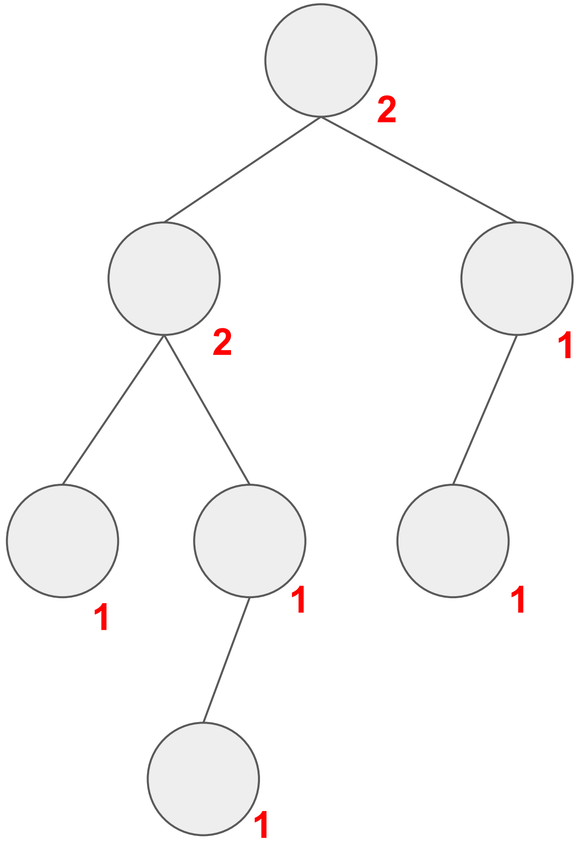


Árvores Esquerdistas

➢ Uma árvore é **esquerdista** se

dist(esq(*x*)) >= *dist*(dir(*x*))

para todo os nós *x* (*dist(null)* = *0*)



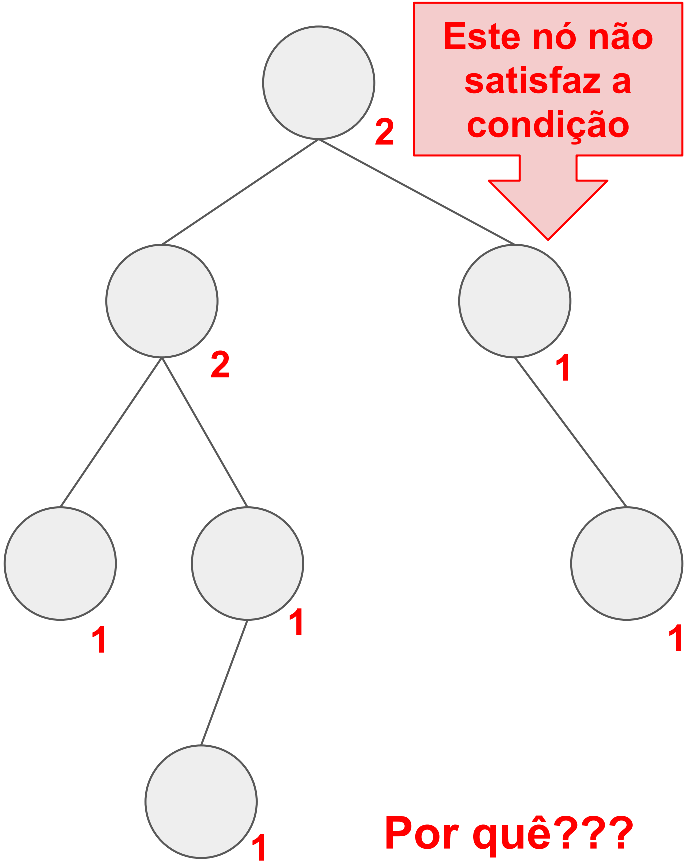
Exemplo 1: Árvore Esquerdista

Heap Esquerdistas

➢ Uma árvore é **esquerdista** se

$$dist(\mathrm{esq}(x)) >= dist(\mathrm{dir}(x))$$

para todo os nós x ($dist(null) = 0$)



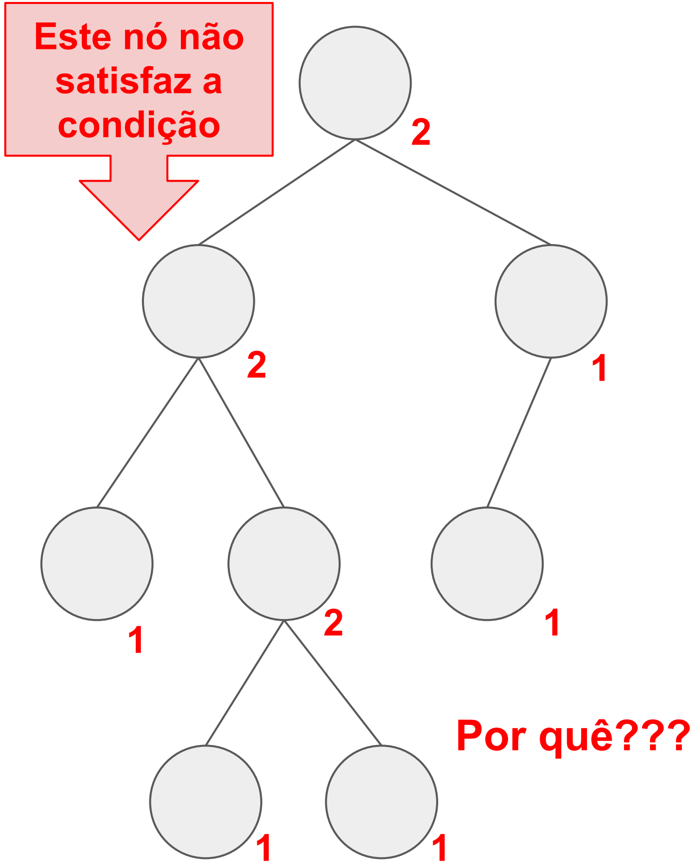
Exemplo 2: Árvore Não Esquerdista

Heap Esquerdistas

➢ Uma árvore é **esquerdista** se

$dist(\text{esq}(x)) >= dist(\text{dir}(x))$

para todo os nós x ($dist(null) = 0$)



Exemplo 3: Árvore Não Esquerdista

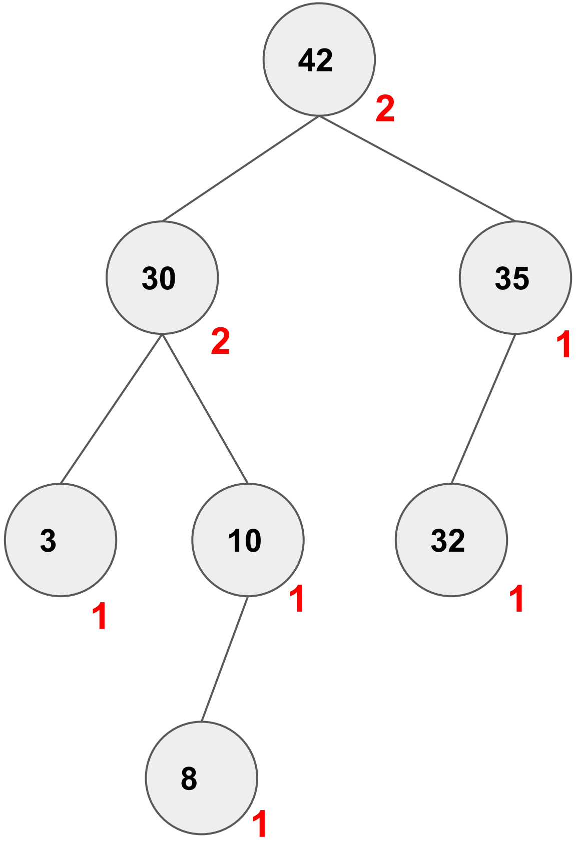
Heap Esquerdistas

Um heap esquerdistas corresponde um conjunto de valores numéricos, denominados prioridade, que são associados aos nós da estrutura T, satisfazendo duas condições:

(1) *Se v é o pai de w em T, a prioridade de v é maior ou igual à de w.*

~~(2) *T é uma árvore binária completa, em que todos os nós do seu último nível se encontra mais à esquerda possível.*~~

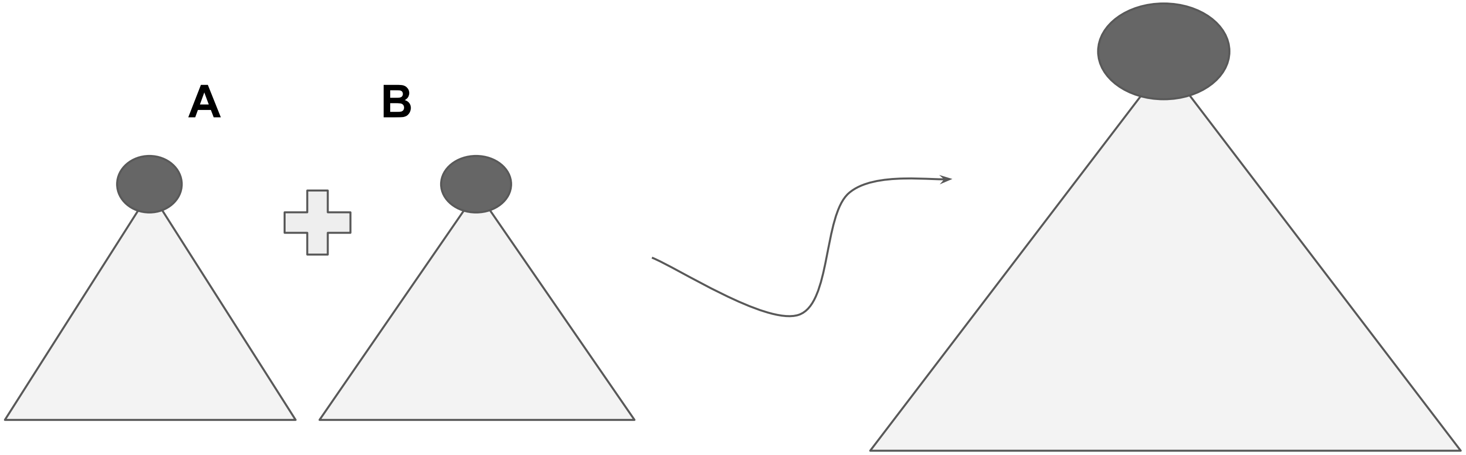
(2) *T é uma árvore binária esquerdista.*



Heap Esquerdista com prioridades

Operação de União

➢ O procedimento de fusão entre duas heap esquerdistas, A e B, retorna uma heap esquerdista que contém a união dos elementos de A e B.



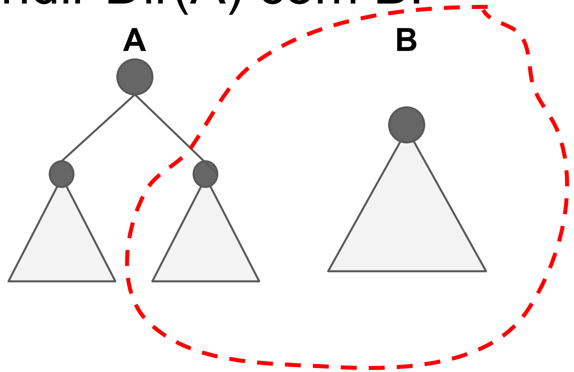
Operação de União

- ➢ Na fusão, o caso mais simples ocorre quando uma árvore está vazia (ou seja, o ponteiro para a raiz é NULL). Nesse caso, basta devolver o outro.

- ➢ Caso contrário, qual é a raiz da árvore mesclada?
  - ○ É a raiz de A se A tiver a chave com maior prioridade ou a raiz de B se tiver maior prioridade.

Operação de União

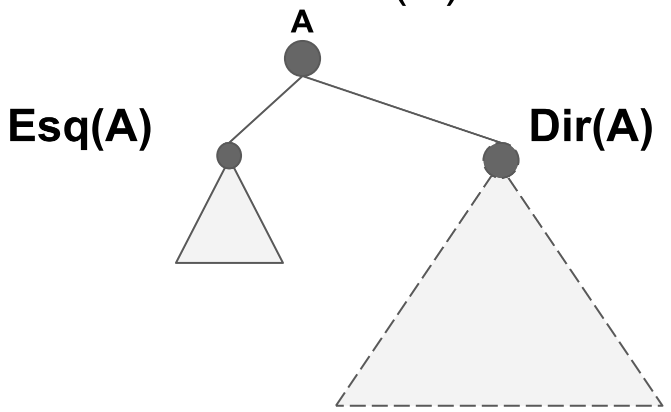
➢ Suponha que a raiz de A tenha a chave com maior prioridade. (Se este não for o caso, simplesmente troque os ponteiros A e B.) Então, podemos fundir Dir(A) com B:



`Dir(A) ← Merge(Dir(A), B)`

Operação de União

- Agora Dir(A) mudou. Seu dist pode ter aumentado no processo, violando a segunda propriedade de heap esquerdistas. Em caso afirmativo, basta trocar à Dir(A) com Esq(A):



`Se (dist(Dir(A)) > dist(Esq(A)) Então troque Esq(A) com Dir(A)`

Operação de União

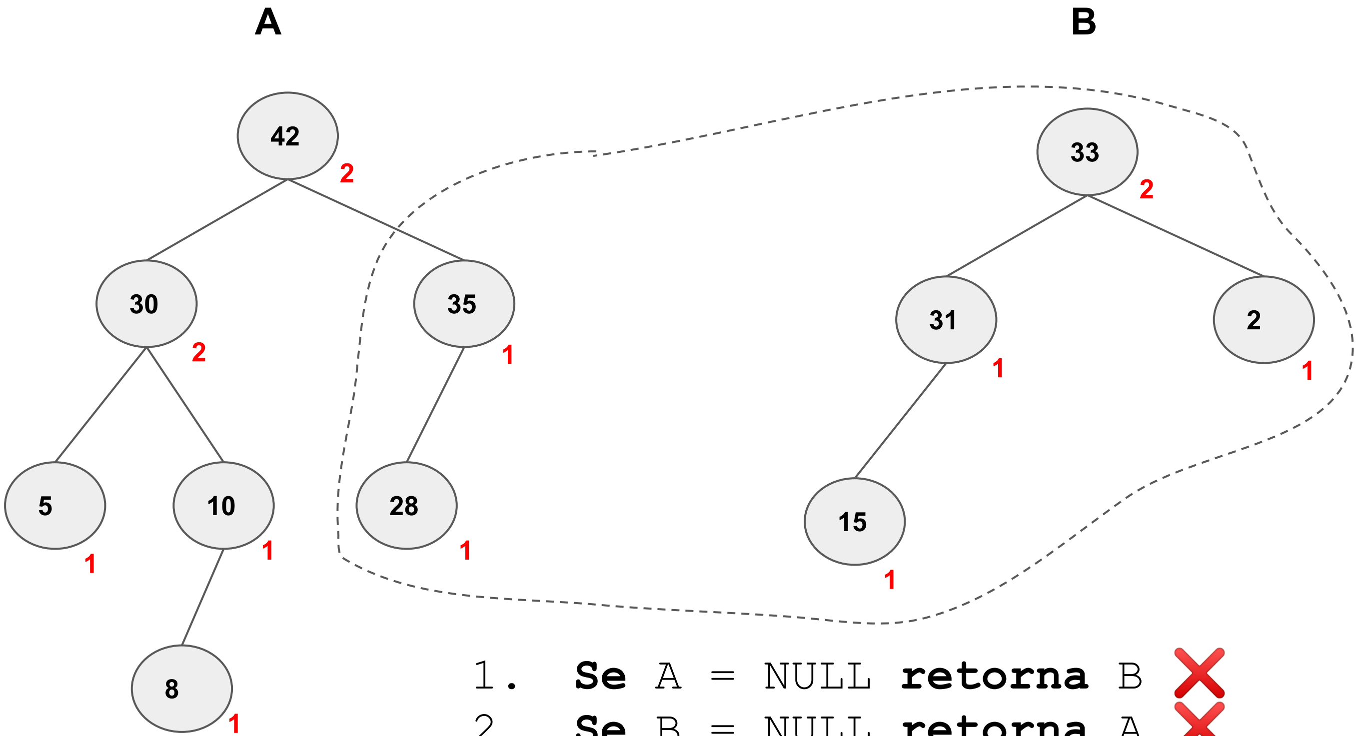
- Finalmente, como dist(dir (A)) pode ter mudado, temos que atualizar dist(A):

```
dist(A) = 1 + dist(dir(A))
```

Algoritmo

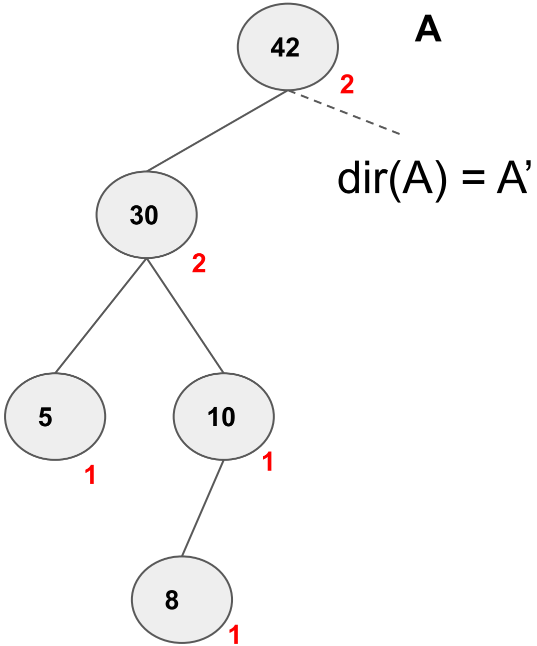
```
Merge(A,B)
Se A = NULL retorna B
Se B = NULL retorna A
Se Chave(B) > Chave(A) Então
      troca (A,B)
Fim-Se
Dir(A) = Merge(Dir(A),B))
Se dist(Dir(A)) > dist(Esq(A) ) Então
    troca (Dir(A), Esq(A) )
Fim-Se
Se Dir(A) = NULL Então
      dist(A) = 0
Caso Contrário
    dist(A) = 1 + dist(Dir(A))
Fim-Se
retorna A
```

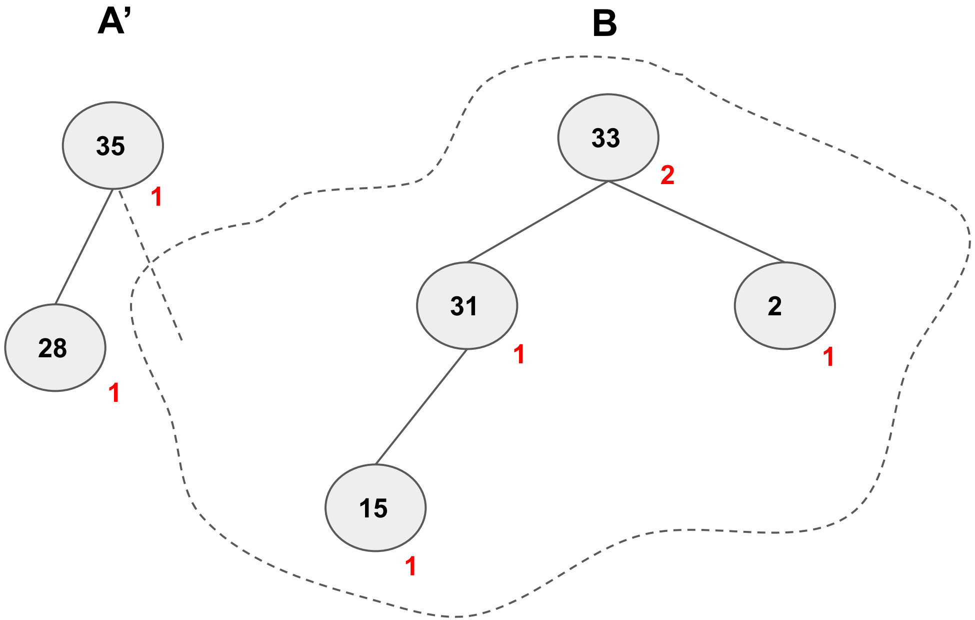
Merge(A,B)



1. **Se** A = NULL **retorna** B ❌
2. **Se** B = NULL **retorna** A ❌
3. Chave(B) > Chave(A) ❌
4. Dir(A) = **Merge(Dir(A),B))**

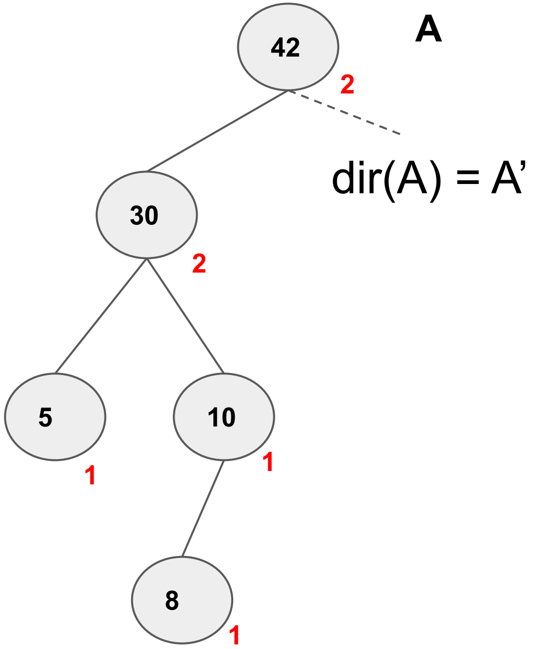
Merge(dir(A),B)

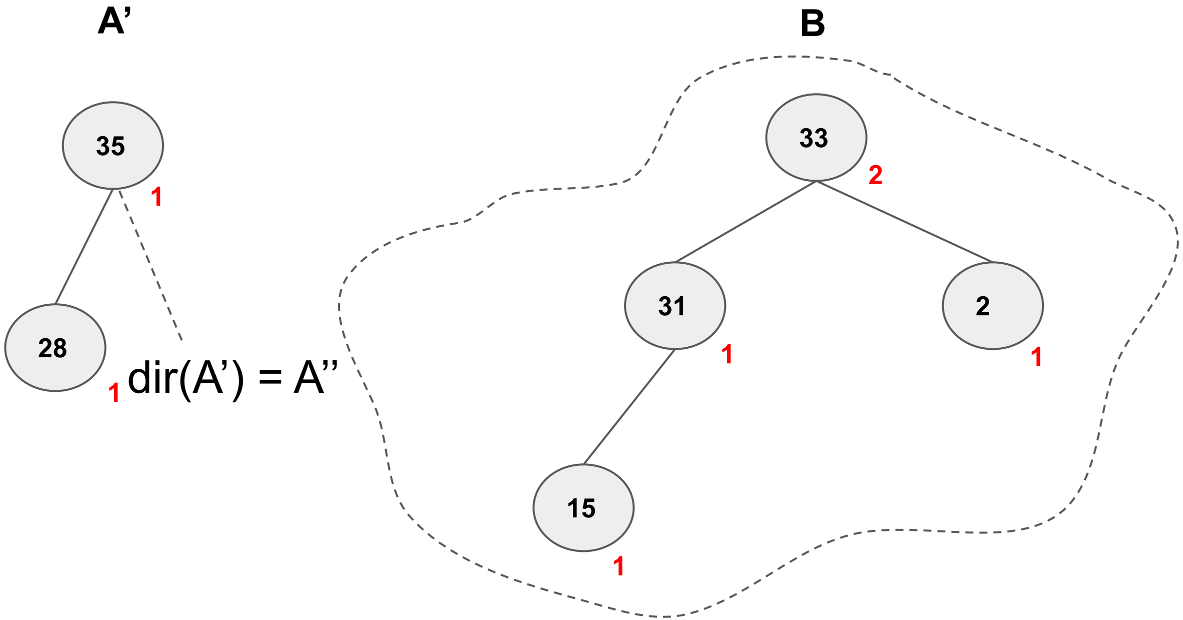




1. **Se** A′ = NULL **retorna** B ❌
2. **Se** B = NULL **retorna** A′ ❌
3. Chave(B) > Chave(A′) ❌
4. Dir(A′) = **Merge(Dir(A′),B))**

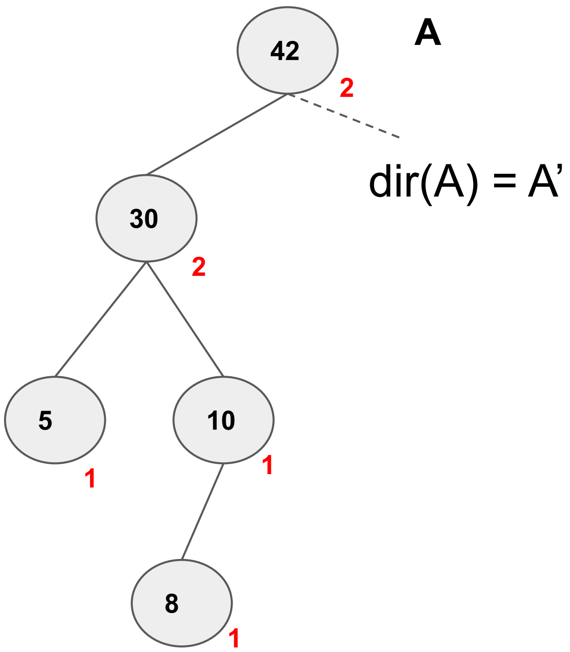
Merge(dir(A'),B)

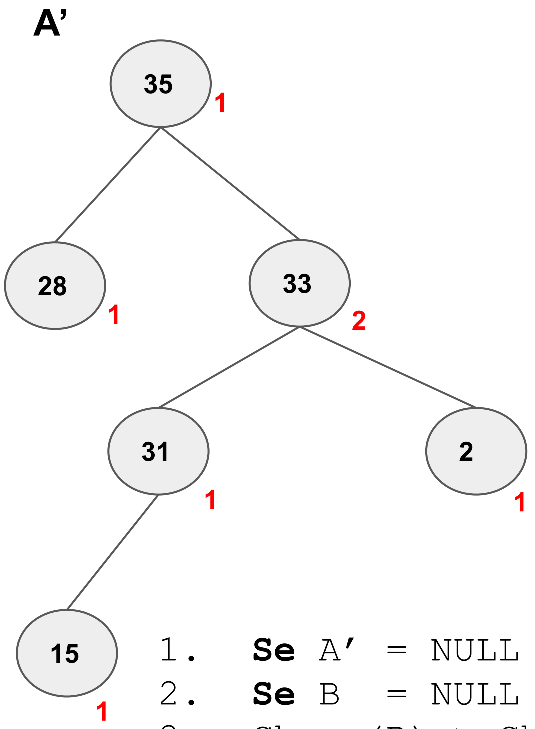




1. **Se** `A'' = NULL` **retorna** `B` ✅

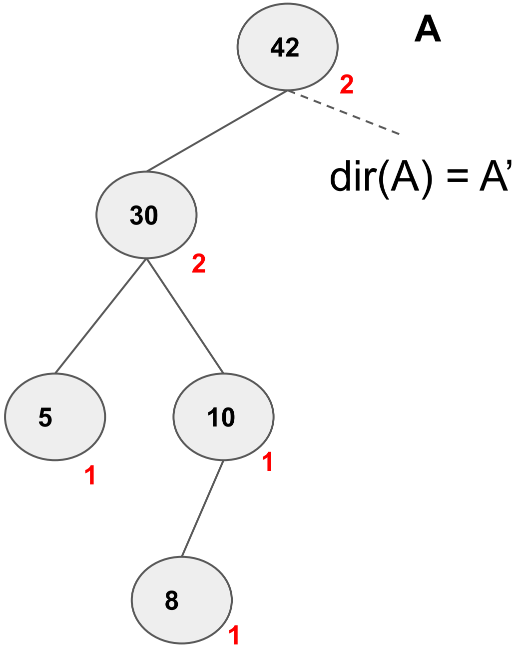
Merge(dir(A),B)

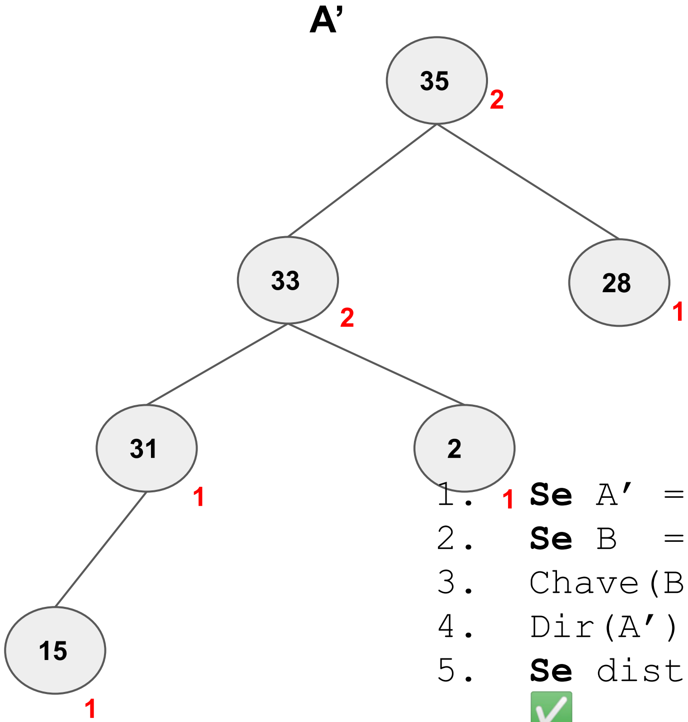





```
1.   Se A' = NULL retorna B ❌
2.   Se B  = NULL retorna A ❌
3.   Chave(B) > Chave(A)    ❌
4.   Dir(A') = Merge(Dir(A'),B))
5.   Se dist(Dir(A')) > dist(Esq(A') ) ✅
6.      troca (Dir(A'), Esq(A') )
```

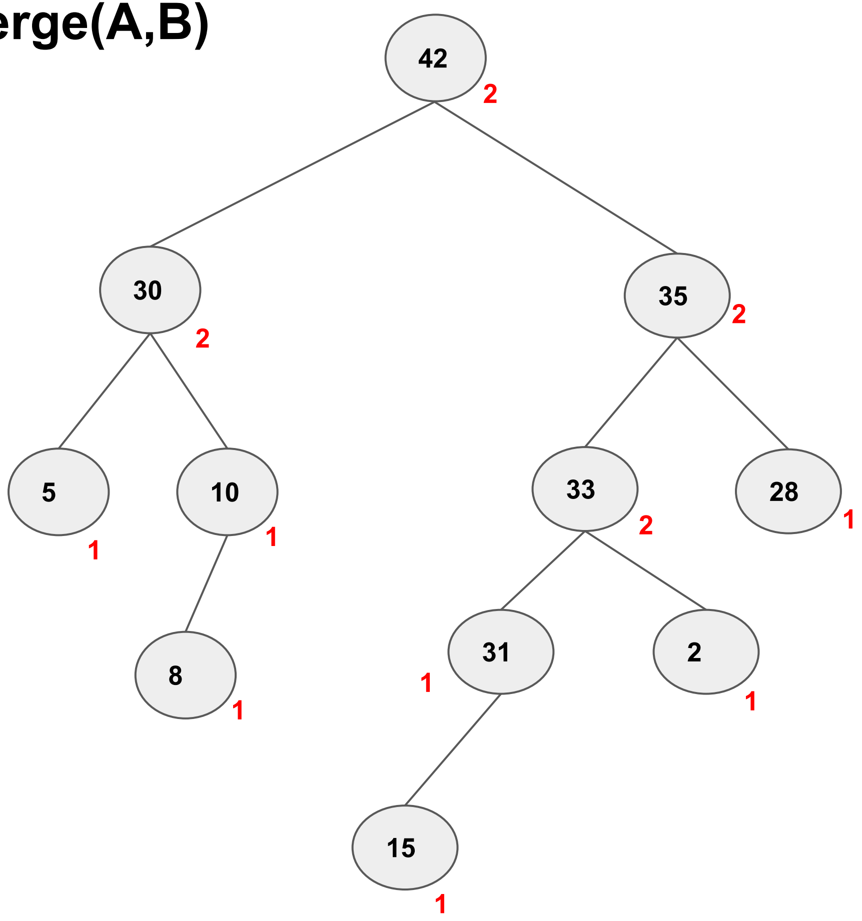
Merge(dir(A),B)





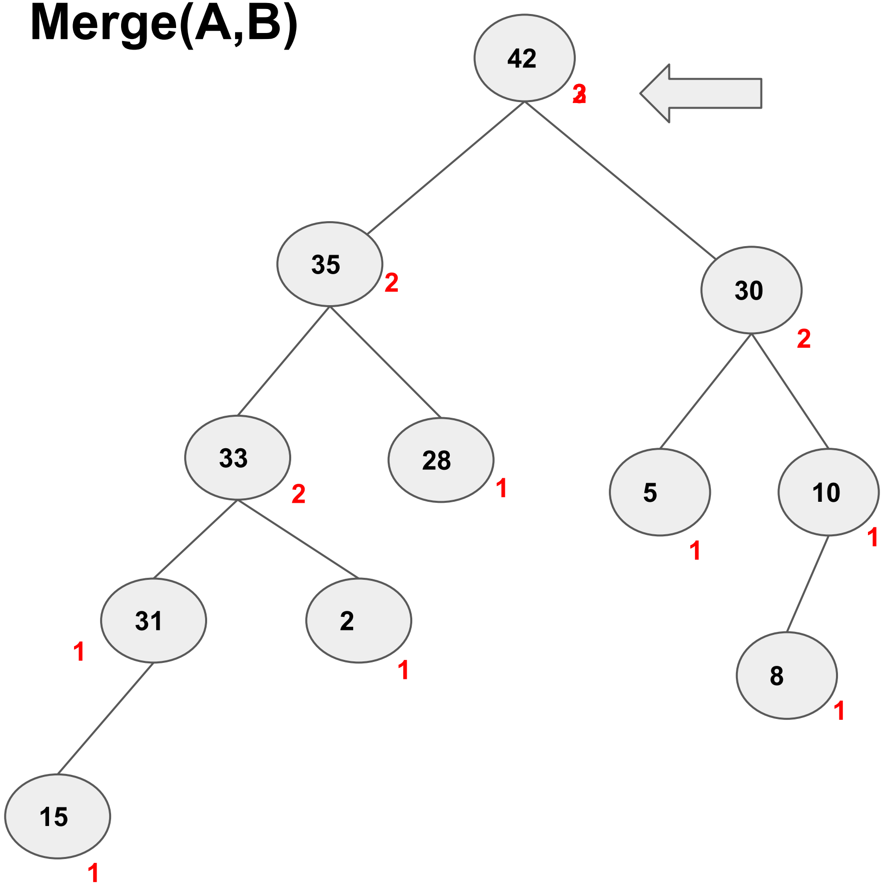

```
Se A' = NULL retorna B ❌
Se B  = NULL retorna A ❌
Chave(B) > Chave(A)    ❌
Dir(A') = Merge(Dir(A'),B))
Se dist(Dir(A')) > dist(Esq(A') ) ✅
    troca (Dir(A'), Esq(A') )
Se Dir(A') = NULL Então
     dist(A') = 0
Caso Contrário
    dist(A') = 1 + dist(Dir(A'))
```

Merge(A,B)




```
Se A = NULL retorna B
Se B  = NULL retorna A
Chave(B) > Chave(A)
Dir(A) = Merge(Dir(A),B))
Se dist(Dir(A)) > dist(Esq(A) )
   troca (Dir(A'), Esq(A') )
Se Dir(A) = NULL Então
    dist(A) = 0
Caso Contrário
   dist(A) = 1 + dist(Dir(A))
retorna A
```

Merge(A,B)

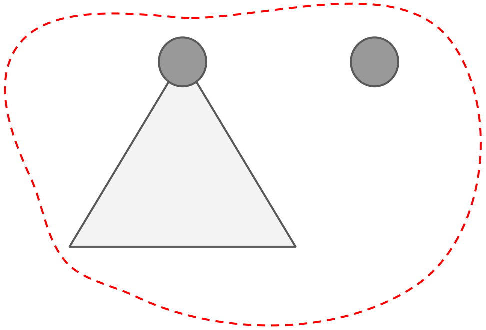



```
Se A = NULL retorna B
Se B  = NULL retorna A
Chave(B) > Chave(A)
Dir(A) = Merge(Dir(A),B))
Se dist(Dir(A)) > dist(Esq(A) )
   troca (Dir(A'), Esq(A') )
Se Dir(A) = NULL Então
    dist(A) = 0
Caso Contrário
   dist(A) = 1 + dist(Dir(A))
retorna A
```

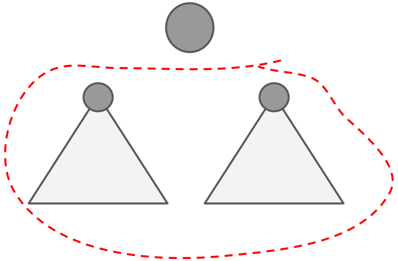
Operações Suportadas com Heap Esquerdistas

- **Inserção**: crie uma heap com único nó e mescla com a heap
- **RemoveMax**: remove o raiz da heap e mescla os filhos

```
Insere(A,x)
    B = CrieHeapEsquerdista(x)
    Merge(A,B)
```



```
RemoveMax(A)
    r = raiz(A)
    Merge(esq(A),dir(A))
    retorne r
```



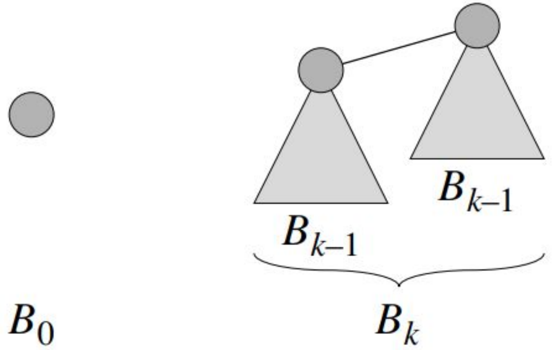
Heaps Binomiais

Árvores Binomiais

➢ Uma árvore binomial de ordem K é uma árvore ordenada definida recursivamente.

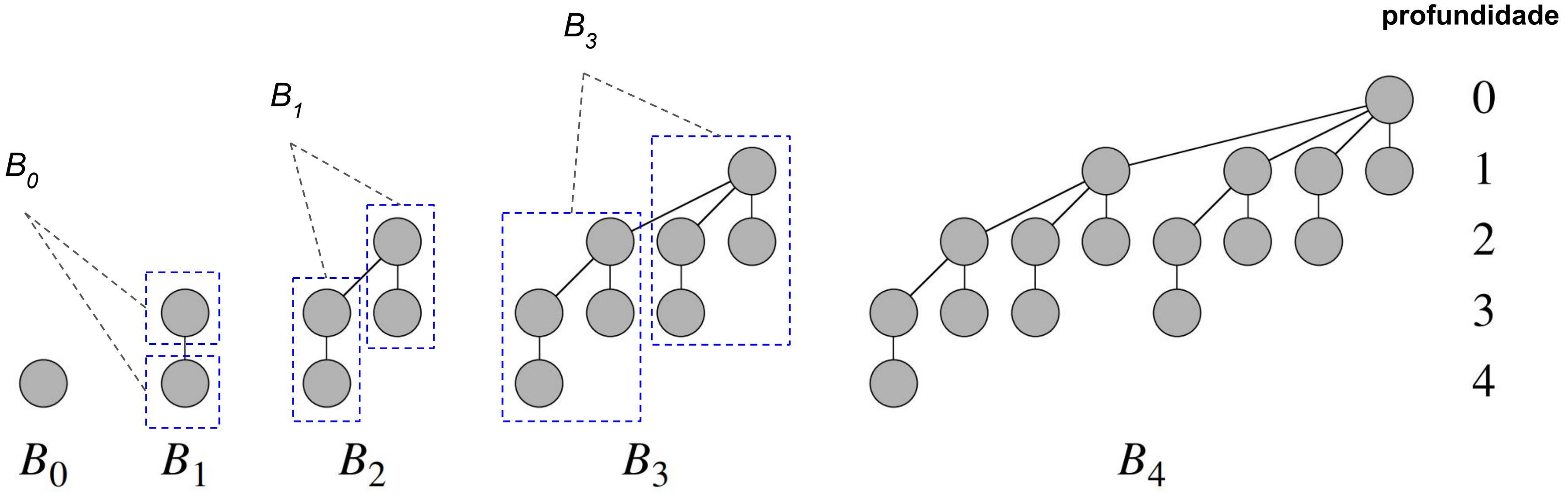
A definição recursiva é fornecida abaixo.

- A árvore binomial de ordem 0, ou seja, k = 0 é um único nó.
- A árvore binomial de ordem k são as duas árvores binomiais de ordem k − 1 ligadas entre si: a raiz de uma é o filho mais à esquerda da raiz de outra.



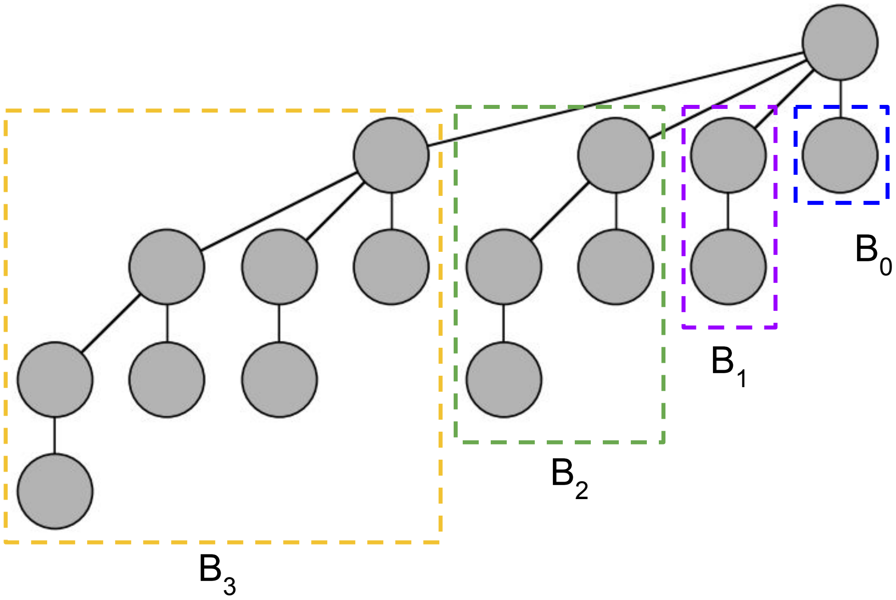
Árvores Binomiais

Exemplos:



Árvores Binomiais

Alternativamente, uma árvore binomial de ordem k é uma árvore cujos filhos são as árvores binomiais de ordem k − 1, k − 2, ..., 1,0



Propriedades da Árvore Binomial

- Para uma árvore binomial de ordem k:
  - Existem 2^k nós
  - A altura da ordem é k.
  - Existe exatamente $\binom{k}{i}$ nós na profundidade *i* (para *i =0, 1,...,k*)
  - Se o nó raiz é deletado, obtém-se as árvores binomiais B_{k-1}, B_{k-2}, …, B_1 e B_o.

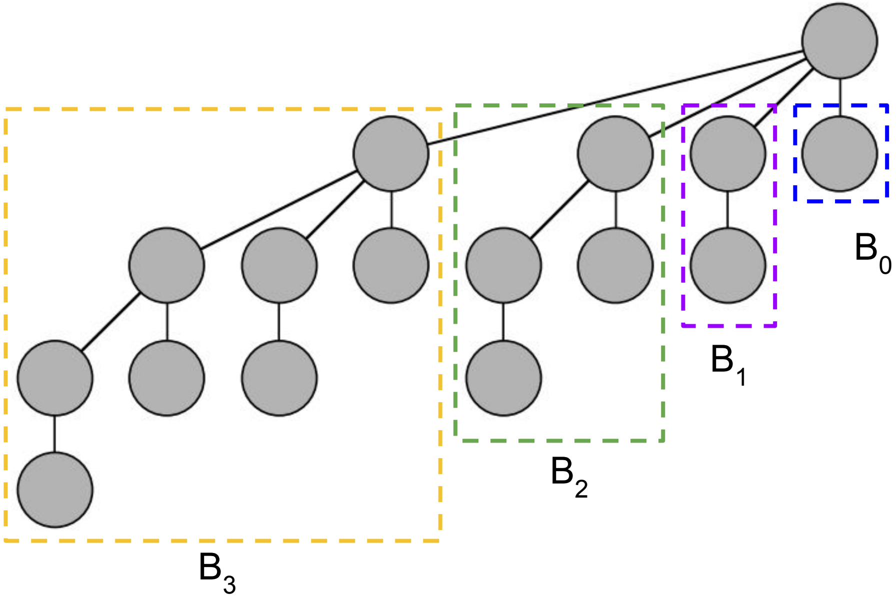
Propriedade de Árvores Binomiais

Dada a árvore binomial de ordem 4, temos:

- ❑ $2^4 = 16$ nós
- ❑ Altura 4
- ❑ Número de elementos na Na profundidade 2 e 3:

$$\binom{4}{2} = \frac{4!}{2!2!} = \frac{4.3.2!}{2.1.2!} = \frac{4.3}{2} = 6$$

$$\binom{4}{3} = \frac{4!}{3!1!} = \frac{4.3!}{3!.1} = \frac{4}{1} = 4$$



Heap Binomial

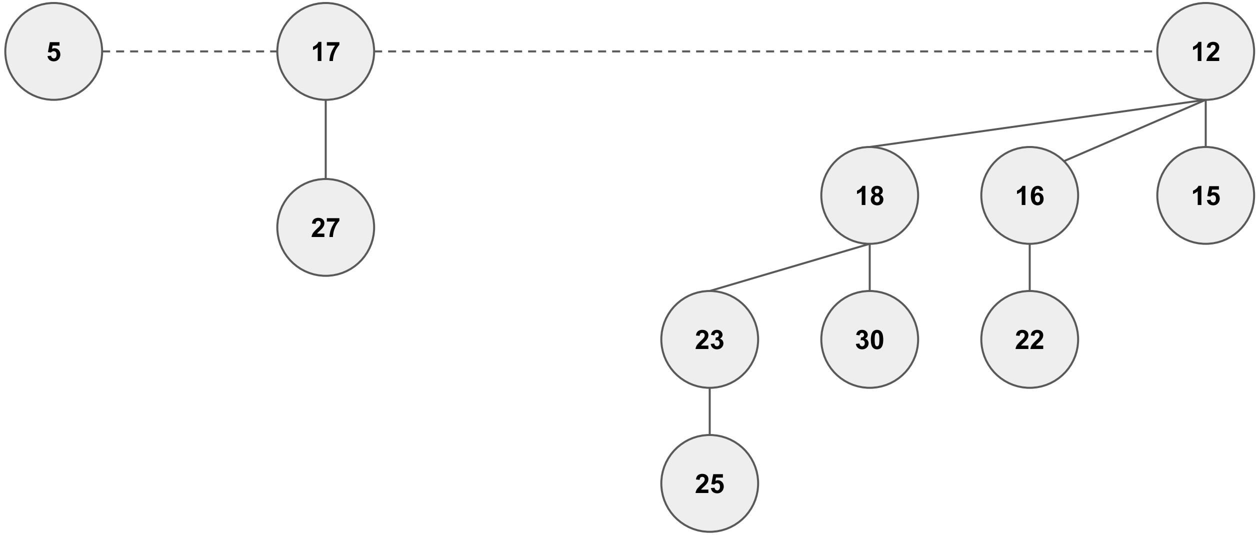
Um heap binomial H é uma coleção de árvores binomiais que satisfaz as seguintes propriedades:

1. Cada nó tem uma chave
2. Cada árvore binomial em um heap binomial de mínimo obedece à propriedade de heap mínimo (que a chave de um nó é maior ou igual à chave de seu pai) e cada árvore binomial em um heap binomial de máximo obedece a propriedade de heap máximo (que a chave de um nó é menor ou igual à chave de seu pai).
3. Para qualquer inteiro não negativo k, há no máximo uma árvore binomial em H cuja raiz tem grau k.

Heap Binomial

Observações:

1. A segunda propriedade nos diz que a raiz de uma árvore ordenada por **heap de mínimo** contém a menor chave da árvore. Se houver m árvores, então a menor chave pode ser encontrada no tempo O(m).



Heap Binomial

Observações:

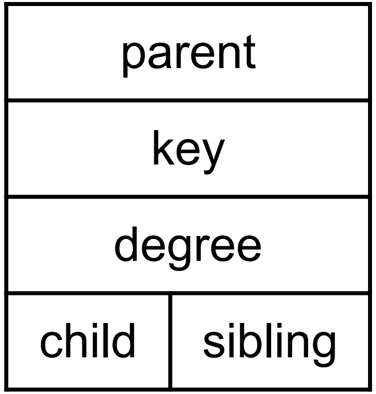
1. A terceira propriedade implica que um heap binomial H de n nós consiste em no máximo $\lfloor \log n \rfloor + 1$ árvores binomiais.

Heap Binomial - Implementação

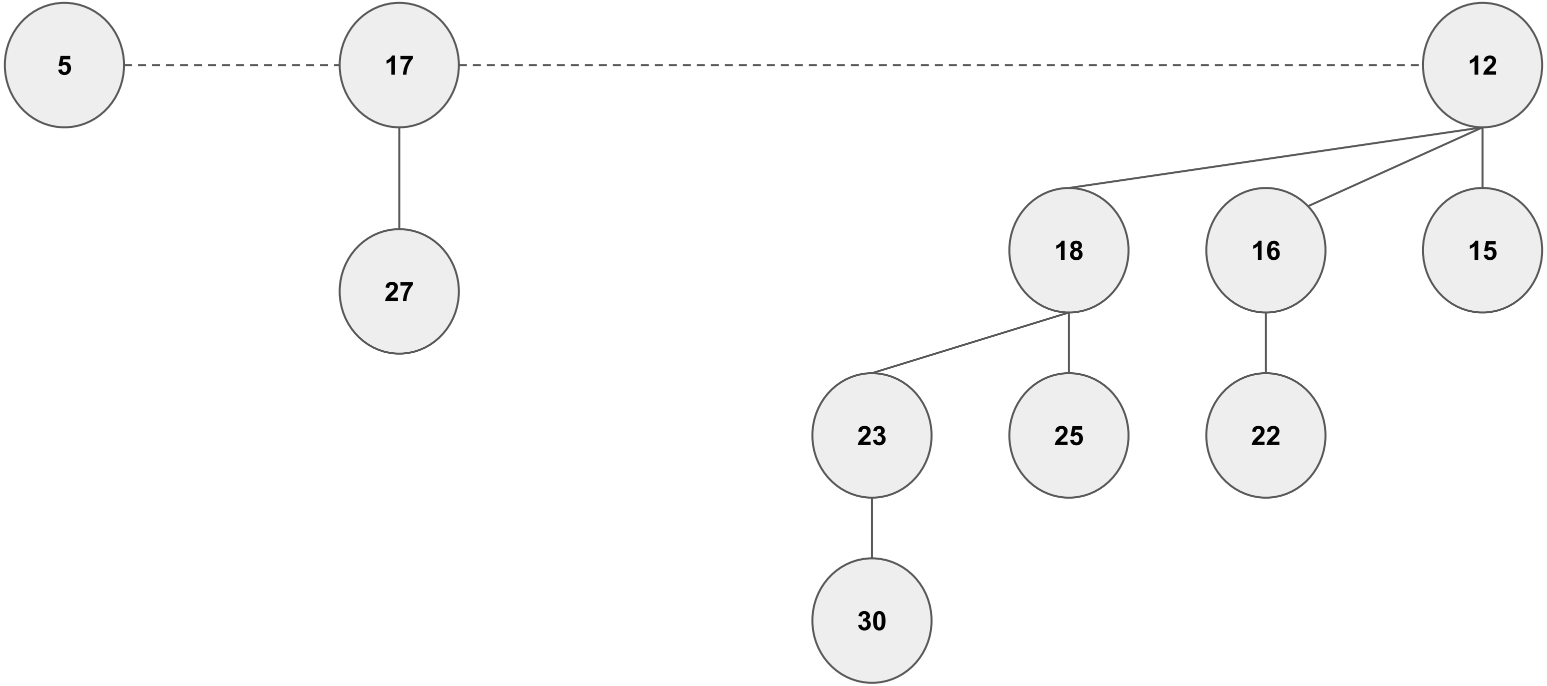
Os heaps binomiais podem ser implementados usando listas vinculadas para armazenar os nós raiz.

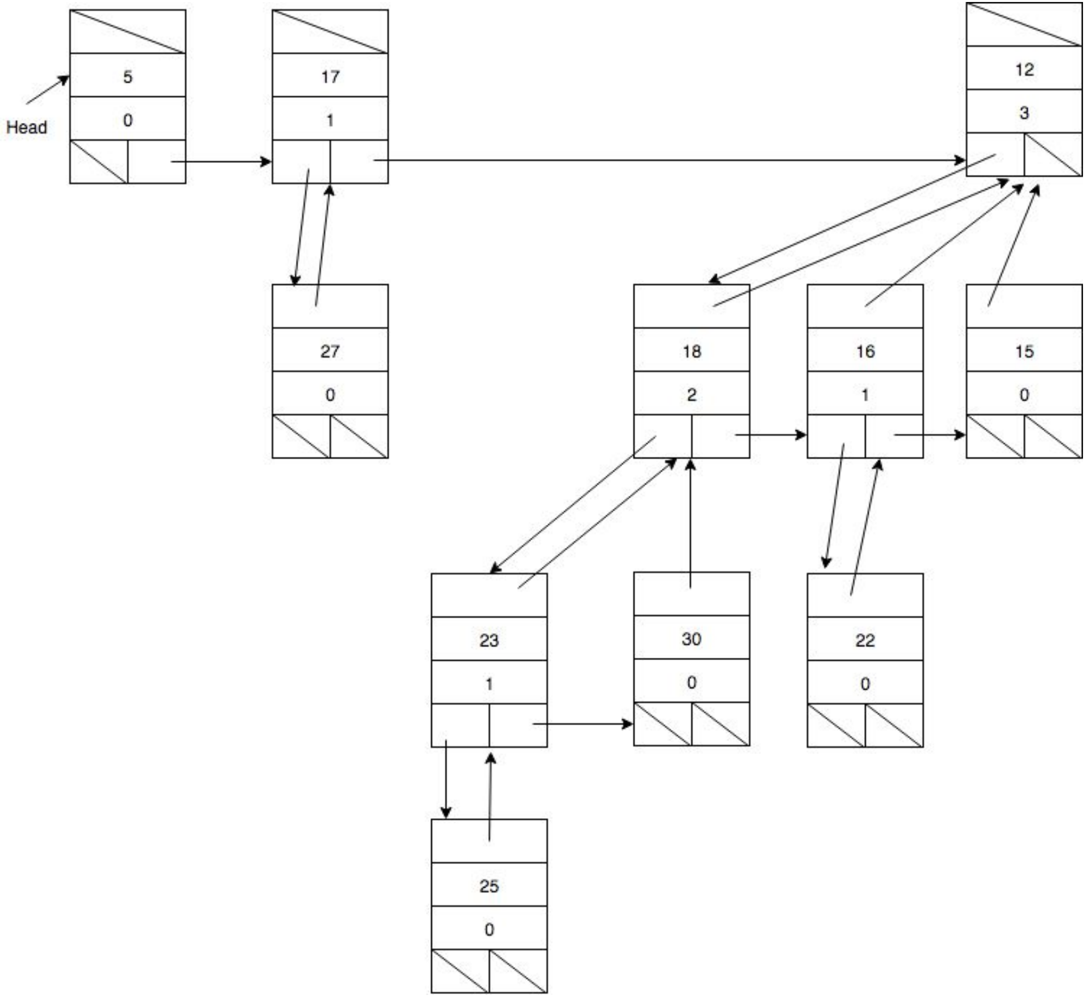
Cada nó armazena as seguintes informações:

- `parent` : ponteiro para nó pai,
- `sibling` : ponteiros para o irmão direito,
- `child` : ponteiro para o filho mais à esquerda,
- `degree` : número de filhos que possui,
- `key` : sua chave.



Heap Binomial - Implementação




Head
5
0
17
1
12
3
27
0
18
2
16
1
15
0
23
1
30
0
22
0
25
0

Operando com Heap Binomial

- **Criando uma nova heap (Make-Heap)**
  - Para criar uma novo heap apenas alocamos e retornamos um estrutura H tal que `head[H] = NIL`.
  - Esta operação tem complexidade O(1).

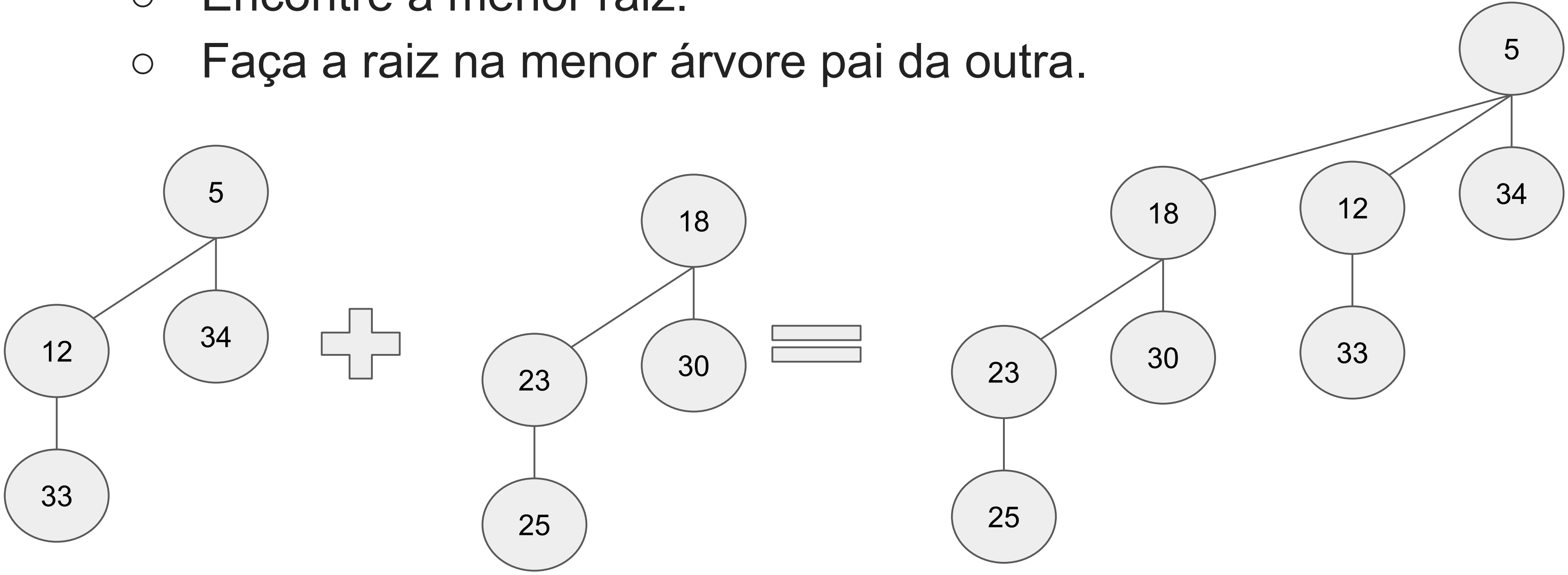
Operando com Heap Binomial

- **Encontrar o mínimo**
  - Retorna um ponteiro para o nó com a menor chave em um heap binomial H com n nós.
  - Complexidade: O(logn).

```
BINOMIAL-HEAP-MIN(H)
    y ← NULL
    x ← head[H]
    min ← ∞
    Enquanto x != NULL Faça
        Se key[x] < min
    Então
            min ← key[x]
            y ← x
        Fim-Se
        x ← sibling[x]
    Fim Enquanto
    retorna y
```

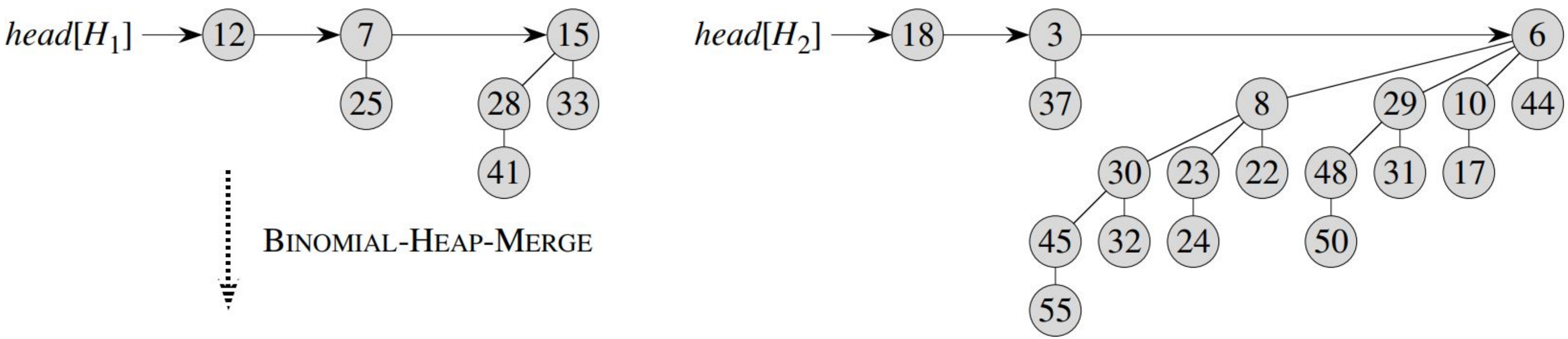
Operando com Heap Binomial

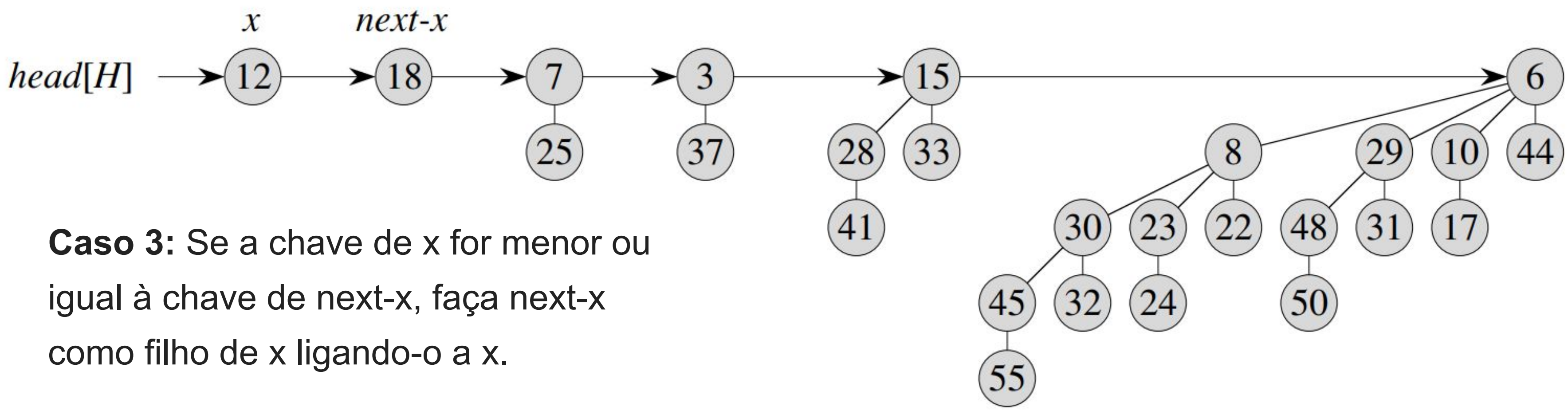
- **Unindo dois heaps binomiais de mesma ordem**
  - Compare as raízes de duas árvores (x e y).
  - Encontre a menor raiz.
  - Faça a raiz na menor árvore pai da outra.



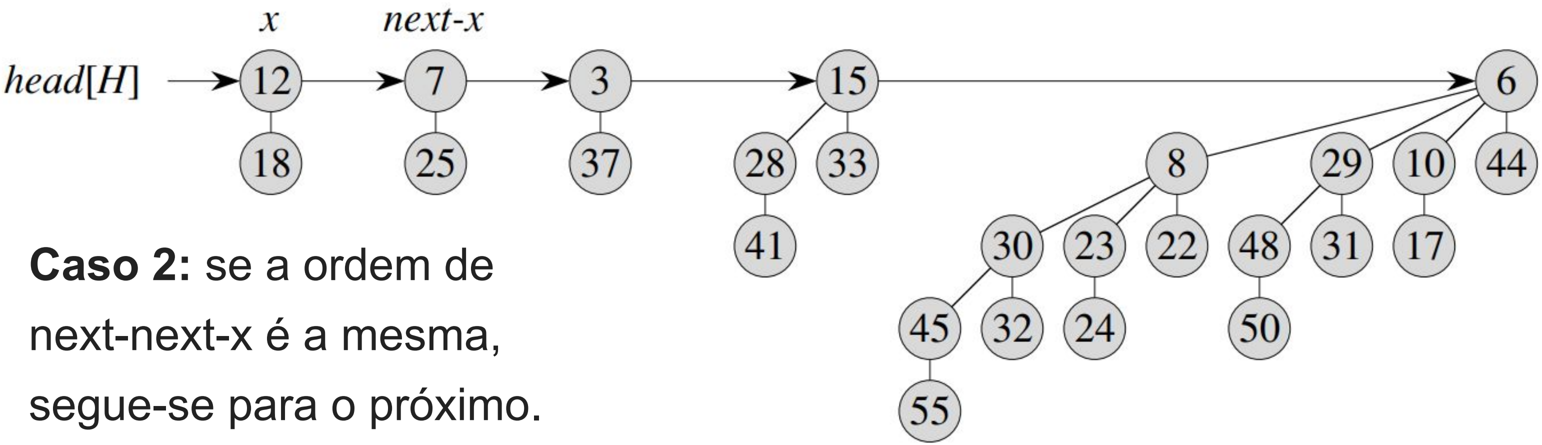
Operando com Heap Binomial

- **Unindo dois heaps binomiais**
  - Junte duas help binomiais sem se preocupar se as árvores são da mesma ordem, colocando-as em ordem crescente de grau.
  - Começando pela cabeça da lista, mescle repetidamente as árvores com o mesmo grau até que todas as árvores na lista tenham um grau único. Dado um nó x e os ponteiros prev-x, e next-x, os seguintes casos podem ocorrer:
    - Caso 1: se a ordem de x e next-x não são iguais, move-se para o próximo.
    - Caso 2: se a ordem de next-next-x é a mesma, segue-se para o próximo.
    - Caso 3: Se a chave de x for menor ou igual à chave de next-x, faça next-x como filho de x ligando-o a x.
    - Caso 4: Se a chave de x for maior, faça x como o filho de next-x.

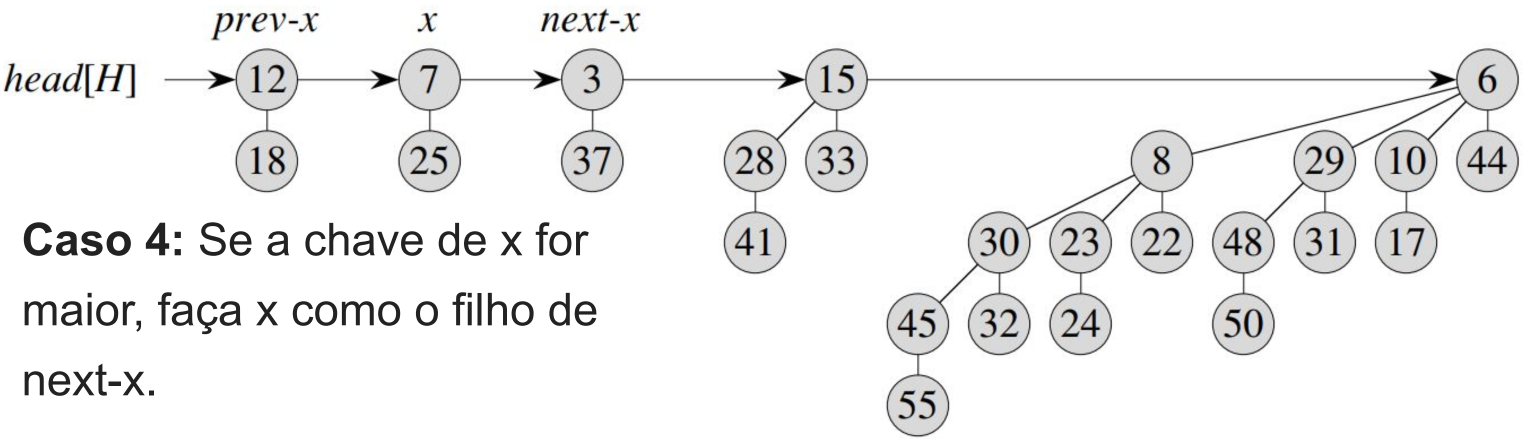




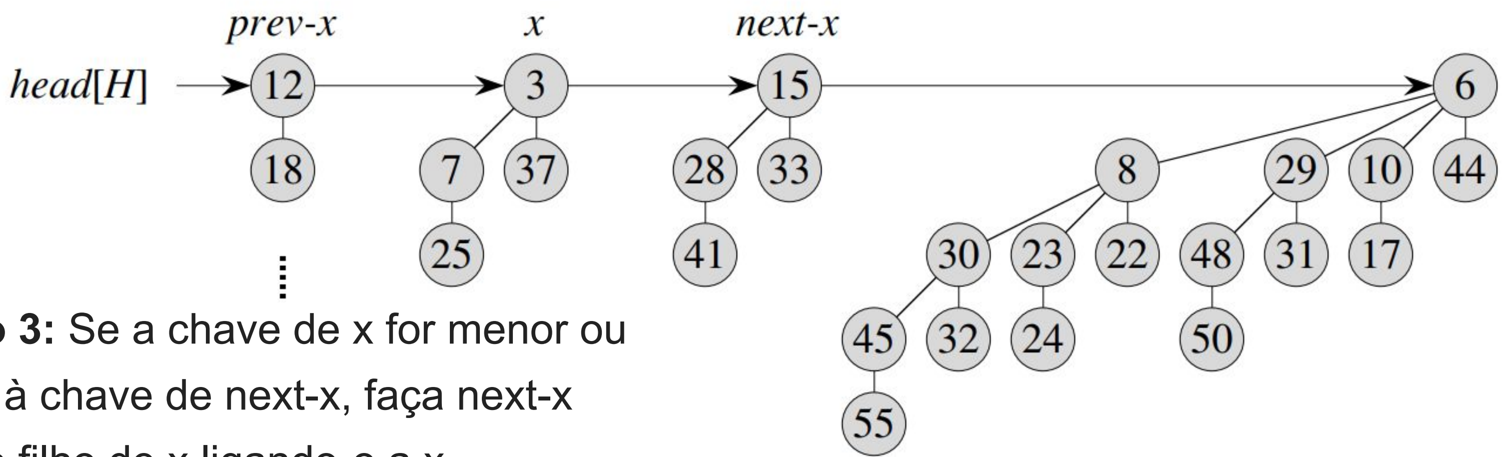
Caso 3: Se a chave de x for menor ou igual à chave de next-x, faça next-x como filho de x ligando-o a x.



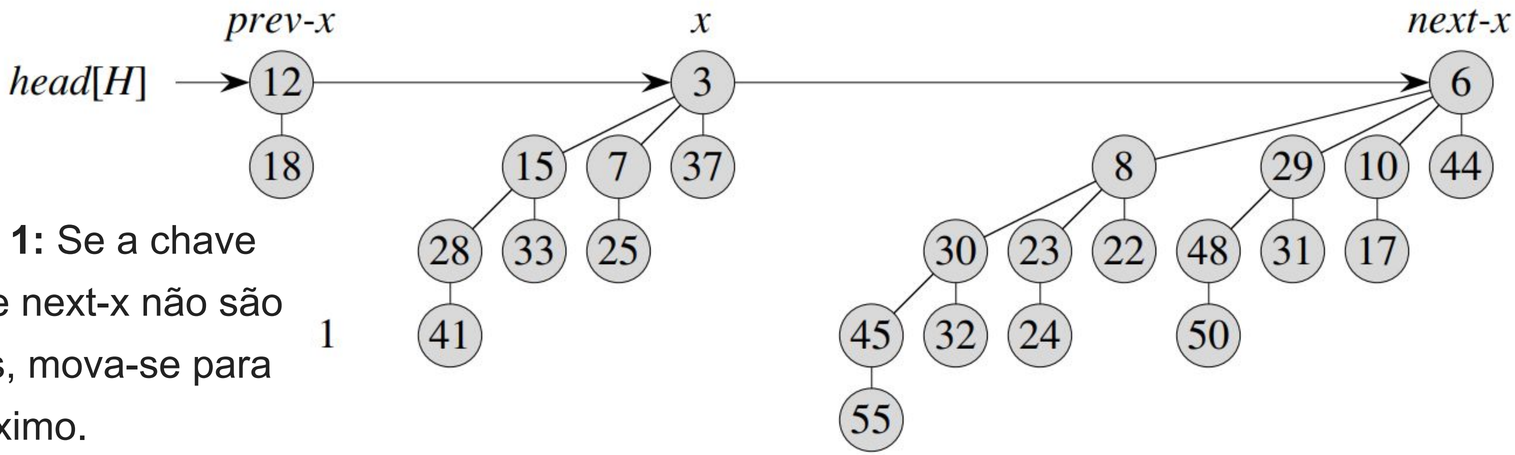
Caso 2: se a ordem de next-next-x é a mesma, segue-se para o próximo.



Caso 4: Se a chave de x for maior, faça x como o filho de next-x.



Caso 3: Se a chave de x for menor ou igual à chave de next-x, faça next-x como filho de x ligando-o a x.



Caso 1: Se a chave de x e next-x não são iguais, mova-se para o próximo.

```
BINOMIAL-HEAP-UNION(H1, H2)
H ← MAKE-BINOMIAL-HEAP()
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
    then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
      do if (degree[x] ≠ degree[next-x]) or
               (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
            then prev-x ← x                                  ▷ Cases 1 and 2
                 x ← next-x                                  ▷ Cases 1 and 2
            else if key[x] ≤ key[next-x]
                    then sibling[x] ← sibling[next-x]        ▷ Case 3
                         BINOMIAL-LINK(next-x, x)            ▷ Case 3
                    else if prev-x = NIL                     ▷ Case 4
                            then head[H] ← next-x            ▷ Case 4
                            else sibling[prev-x] ← next-x    ▷ Case 4
                         BINOMIAL-LINK(x, next-x)            ▷ Case 4
                         x ← next-x                          ▷ Case 4
         next-x ← sibling[x]
return H
```

BINOMIAL-HEAP-MERGE(H1,H2)
Mescla as heaps H1 e H2 em uma única lista ordenando as raízes em ordem monotonicamente crescente

BINOMIAL-LINK(y,z)
Torna o nó z a raiz da árvore apontada por y no tempo O (1)

```
1. parent[y] ← z
2. sibling[y] ← child[z]
3. child[z] ← y
4. degree[z] ←degree[z]+1
```

A complexidade do algoritmo de união é O(lg n)

Operando com Heap Binomial

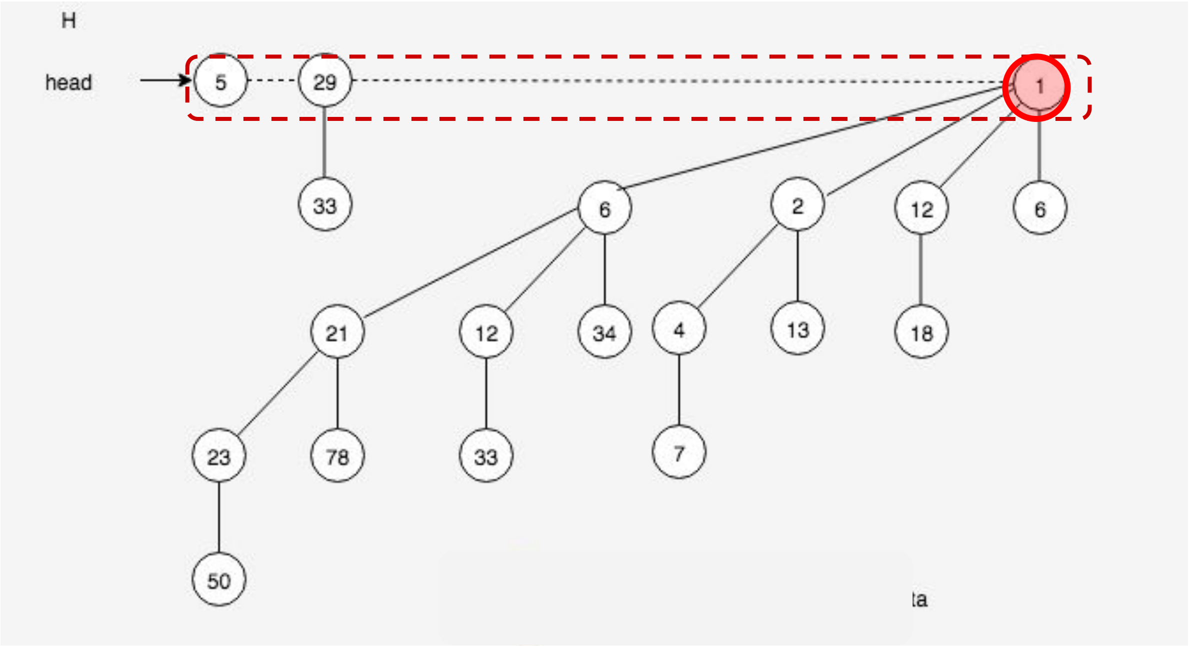
- **Inserindo um novo nó na heap**
  - Para inserir um nó basta criar uma novo heap contendo apenas este elemento e uni-lo a heap em que queremos inseri-lo.
  - Complexidade: O(log n)

```
Binomial-Heap-Insert(H, x)
1. H’ ← Make-Binomial-Heap()
2. parent[x] ← NULL
3. child[x] ← NULL
4. sibling[x] ← NULL
5. degree[x] ← 0
6. head[H’] ← x
7. H ← Binomial-Heap-Union(H,
H$)
```

Operando com Heap Binomial

➢ **Extraindo o mínimo da heap**

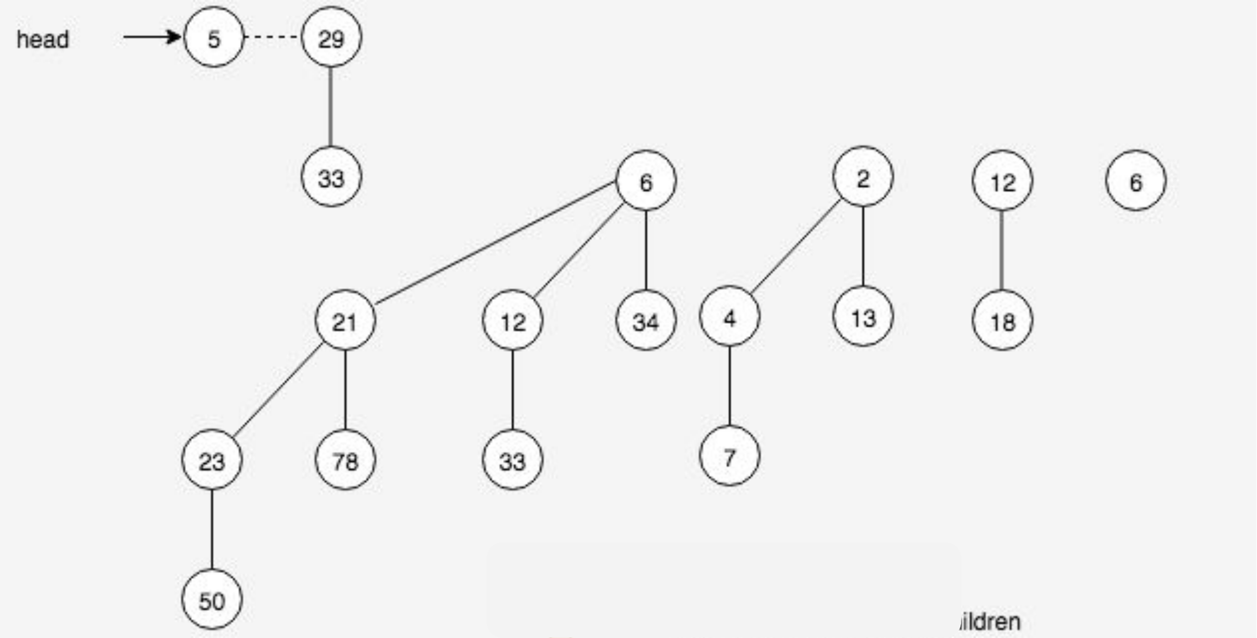
Passo 1: Procure nas raízes das árvores binomiais a raiz com a menor chave



Operando com Heap Binomial

- ➢ **Extraindo o mínimo da heap**

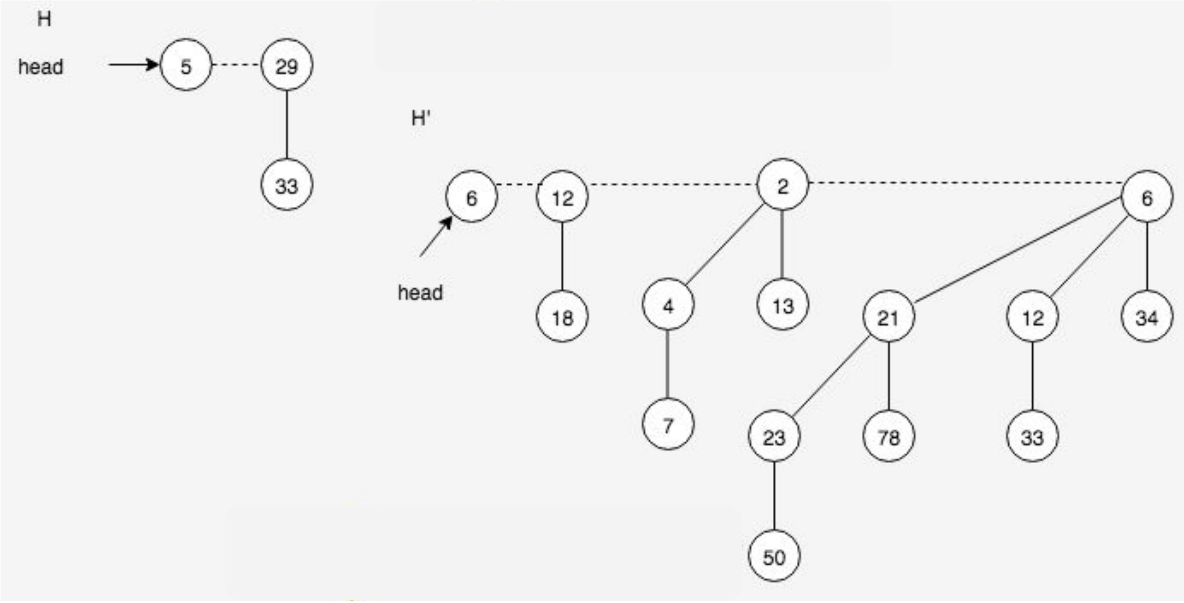
Passo 2: Remova da árvore.



Operando com Heap Binomial

➢ **Extraindo o mínimo da heap**

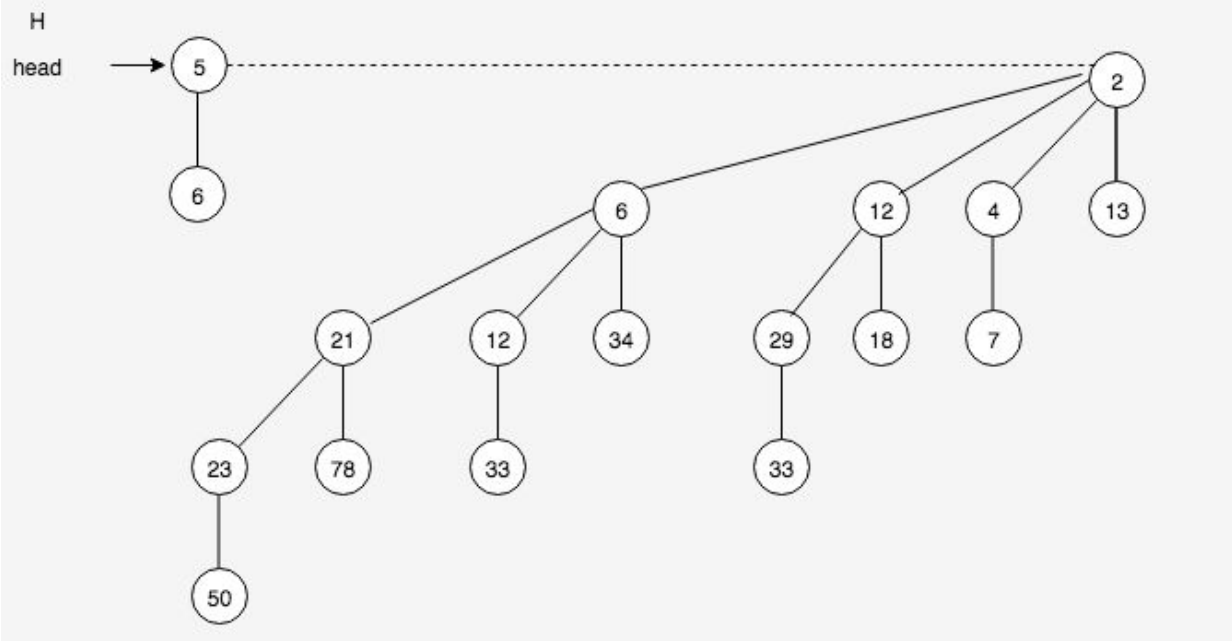
Passo 3: Construa uma lista com a ordem invertida dos filhos do nó removido e defina a raiz de nova heap binomial H' para apontar para a raiz da lista resultante.



Operando com Heap Binomial

➢ **Extraindo o mínimo da heap**

Passo 4: Unir as Heaps H e H'



Filas de Prioridades - Resumo

Procedimento	**Heap Binário** (pior caso)	**Heap Esquerdistas** (pior caso)	**Heap Binomial** (pior caso)	**Heap Fibonacci** (amortizado)
Make-Heap	O(1)	O(1)	O(1)	Θ(1)
Insert	Θ(logn)	O(logn)	O(logn)	Θ(1)
Min/max	O(1)	O(1)	O(logn)	Θ(1)
Extract-Min/max	Θ(logn)	O(logn)	Θ(logn)	O(log n)
Union	Θ(n)	O(logn)	O(logn)	Θ(1)